A RAND NOTE

Dependencies, Demons, and Graphical Interfaces in the Ross Language

Stephanie Cammarata, Barbara Gates,
Jeff Rothenberg

March 1988

40 Years 1948-1988
RAND

The research described in this report was sponsored by the Defense Advanced Research Projects Agency under RAND's National Defense Research Institute, a Federally Funded Research and Development Center supported by the Office of the Secretary of Defense, Contract No. MDA903-85-C-0030.

Published by The RAND Corporation
1700 Main Street, P.O. Box 2138, Santa Monica, CA 90406-2138

A RAND NOTE

N-2589-DARPA

Dependencies, Demons, and Graphical Interfaces in the Ross Language

Stephanie Cammarata, Barbara Gates,
Jeff Rothenberg

March 1988

Prepared for
The Defense Advanced Research Projects Agency

PREFACE

This research was sponsored by the Defense Advanced Research Projects Agency under the auspices of RAND's National Defense Research Institute, a Federally Funded Research and Development Center sponsored by the Office of the Secretary of Defense. The study was conducted in the Information Processing Systems program. Under this sponsorship, RAND has been investigating the utility of knowledge-based artificial intelligence techniques to help solve deficiencies that exist in large-scale military simulations. The research described in this Note is one aspect of this project and addresses simulation consistency issues. RAND's analysis of this problematic area is based on experience gained from the implementation of an object-oriented simulation demonstrating cooperative intelligence for remotely piloted vehicle fleet control.

This Note serves four purposes. First, it explicitly identifies the source of many inconsistency problems in object-oriented simulations. Second, it presents a general methodology the authors have developed for solving three classes of inconsistencies, and the solutions we have implemented as extensions to the Ross object-oriented simulation language. Third, it provides an example simulation written in Ross and Lisp demonstrating the utilization of the extensions. Finally, in an appendix it supplies guidelines to aid a simulation developer in the use of these new capabilities. Although these extensions were developed for the Ross language, these facilities are equally applicable to any *object-oriented message-passing* programming language.

SUMMARY

An object-oriented style of computation is especially well suited to simulation in domains that consist of *intentionally* interacting components. In such domains, the programmer can map the constituent domain components onto objects, and intentional interactions (e.g., communications) onto message transmissions. However, some events or interactions between real world objects cannot be modeled as naturally as we might like. Improper modeling of these interactions inevitably leads to inconsistent simulation states and processing errors.

The research reported in this Note identifies three categories of simulation activities that are unnatural to model and difficult to implement in object-oriented simulations. The three categories include: (1) scheduling events that depend on the continuous aspect of time; (2) simulating non-intentional side effects of deliberate actions; and (3) presenting a graphics display of a simulation so that any changes in the simulation state are immediately visible.

Following a discussion of these categories, we present a methodology for performing these tasks that is transparent to the simulation programmer. Our approach utilizes extensions to the Ross object-oriented language, allowing a programmer to declaratively specify characteristics of the simulation dealing with time-dependent attributes, non-intentional events, and graphics display strategies.

The example presented in this Note demonstrates the many advantages of our declarative approach to maintaining consistency. With these capabilities, we expect object-oriented simulation languages to become increasingly attractive for modeling dynamic systems.

CONTENTS

FIGURES

1. INTRODUCTION

The object-oriented programming paradigm suggests a natural way to model many dynamic systems. Object-oriented languages (Stefik and Bobrow, 1986) encourage a simulation designer to model physical entities by building software objects analogous to real world entities, and endowing those objects with methods and behaviors for responding to model stimuli. The programming tools and techniques supporting object-oriented languages have produced simulation systems that are far more comprehensible and analyzable than those developed in conventional simulation languages.

Since 1982, RAND researchers have been building and using object-oriented languages for military modeling and simulation (McArthur et al., 1984). We have garnered much information about the programming needs of simulation modelers and analysts. Although an object-oriented message-passing language is powerful for simulating discrete, synchronous events, it lacks flexibility for modeling other types of integrated processes in a convenient and natural fashion (Ruiz-Mier and Talavage, 1987). The work described in this Note focuses on techniques for simulating continuous and asynchronous processes in an object-oriented simulation language. The products of this work are prototype extensions to the Ross (Rule-Oriented Simulation System) (McArthur et al., 1985) language for modeling such activities.

In an object-oriented simulation system, the program entities and processes correspond closely to those objects and activities being simulated. Interactions between the various objects are represented as messages that are passed between simulated objects. Nevertheless, many additional computational tasks must be programmed that have no analogy in the modeled world. For instance, in most object-oriented simulations, mobile vehicles such as aircraft or tanks have an associated behavior for prescribing how the vehicle's location should be computed. This computation is most often based on parameters such as the vehicle's previous location, its velocity, and the amount of time that has elapsed since the previous position computation. Clearly, in a live scenario with moving objects, the vehicle's driver never invokes an activity analogous to *computing one's position.* Instead, one's position is always changing (as long as the velocity is greater than zero). Although a pilot may *notice* or *record* an aircraft's current position, the physical characteristics of the scenario result in a continuously changing position. Therefore, in a computer simulation, *updating position* is one kind of computational activity that has no corresponding real-

world analog. We refer to these activities and the simulation code required to model these physical phenomena as *artifactual.* Most simulation languages require a programmer to develop many artifactual procedures, necessary only because of the limitations of our current methods for computer simulation.

The first goal of the work presented in this Note is to identify and categorize some of the artifactual activities that must be maintained by a simulation programmer. The second is to describe some automatic programming aids that reduce the programmer's burden of handling artifactual processes. This methodology enables a programmer to *declaratively* specify the parameters and routines that are necessary for artifactual activities. These declarations are included with the class, subclass, and property declarations localized in the class hierarchy specifications of an object-oriented simulation. This technique eliminates the need for a programmer to *procedurally* manage processes that do not correspond to modeled activities.

In Section 2 we discuss the background and motivation for this work. Section 3 describes the deficiencies and problems of current object-oriented simulation systems this research is focusing on. Declarative programming techniques addressing three problematic situations are presented in Section 4. The three areas include attribute dependencies, asynchronous simulation activities, and approaches for interfacing to a graphical display. Section 5 provides a comprehensive example of the use of these tools. This example is extracted from an existing simulation system; therefore, some familiarity with Lisp and object-oriented programming is helpful. The pictorial figures in Section 5 depict simulation graphics produced on a Sun workstation using the SunCore graphics package (Newman, 1978) and Hose interface facility (Rothenberg, 1987). In Section 6 we present implementation details, and the final section identifies some of the benefits and limitations of this work and concludes with some statements about future directions. Appendix A supplies detailed guidelines for using the Ross extensions in an object-oriented simulation system. In Appendix B, we list the Ross and Lisp source code required for the example detailed in Section 5.

2. BACKGROUND AND MOTIVATION

The development of RAND's object-oriented simulation language, Ross, was motivated from two arenas: the military simulation work conducted by RAND's Project Air Force division, and the advent of object-oriented and message-passing languages such as Smalltalk (Goldberg and Robson, 1982) and Simula (Birtwistle et al., 1973) among artificial intelligence research centers. Because existing simulation models in Fortran proved to be extremely unwieldy in many respects, military analysts were searching for better ways to represent and simulate their complex battle management models (Kornell, 1987). The introduction of object-oriented languages suggested a paradigm that would make these simulations easier to build, use, and understand. In an effort to achieve these goals, Ross was patterned after the Director (Kahn, 1979) system and initially used for the Swirl simulation model (Klahr et al., 1982a). Since 1982, many of RAND's simulation projects have been implemented in Ross, including Twirl (Klahr et al., 1984), Identification: Friend or Foe (Callero et al., 1985), and Distributed Fleet Control (Steeb et al., 1986a and 1986b). In addition, Ross has been distributed to research sites outside of RAND, where it has been used for a variety of applications (Dockery, 1982; Conker et al., 1983; Nugent, 1983; Hilton, 1986).

Ross has achieved its initial objective; however, we and other researchers have observed that more advanced object-oriented simulation tools are desperately needed (Overstreet and Nance, 1985; Elza, 1986; Rothenberg, 1986; Reddy, 1987). The lack of suitable state/time-based constructs in object-oriented simulation languages presents particular difficulty for simulating certain kinds of events. Radiya (Radiya and Sargent, 1987) proposes integrating rules and objects for overcoming this deficiency, and McArthur (McArthur, 1987) addressed this issue in his work utilizing object-oriented simulations as training and tutoring aids. RAND's Knowledge Based Simulation (KBSim) project is devoted to providing an effective development environment for building discrete event object-oriented simulation models (Rothenberg et al., 1987). The work presented in this Note is one aspect of the KBSim project.

Although this work extends the *Ross* simulation language, the methodology we discuss is equally applicable to any object-oriented *simulation* language. We characterize an object-oriented simulation language as an object-oriented programming language augmented with primitive simulation objects, such as a "clock" object. Other simulation primitives

include a "tick" behavior for scheduling future events and processing an object's event queue.

In Ross, both *classes* and *instances* are "objects" because both can send and receive messages. However, our use of class objects for sending and receiving messages is minimized. Our objective was to develop this work so that it could be applied to other object-oriented languages that do not treat classes as objects. In this Note we refer to class objects as *classes* or *generic classes*, and instance objects simply as *objects* or *instances*. Procedural blocks of code known as "methods" in Smalltalk and other object-oriented languages are called *behaviors* in Ross. Therefore, we use the term *behavior* throughout this Note.

3. PROBLEMATIC ARTIFACTUAL ACTIVITIES

As the development and implementation of an object-oriented simulation system evolves, objects and behaviors are continually added and modified. Although an object-oriented message-passing paradigm appears natural and direct, it has, nevertheless, data consistency issues that must be maintained by the programmers. Ignoring these issues will result in incorrect simulation processing and inconsistent results. In this section we detail three categories of simulation processing that, if not handled correctly, result in conflicting simulation computations.

3.1. Maintenance of Continuous Attributes

In general, object-oriented simulations are discrete, event-based models; time is advanced as the result of events scheduled in the future. Therefore, the passage of time is a side effect of the occurrence of events. This event-driven strategy has far-reaching implications for simulation processing. The effects are most noticeable when objects that move continuously with time are simulated. Simulated moving objects do not normally move by themselves; rather, they must have their positions updated whenever some event occurs that advances time. We categorize attributes whose values depend continuously on time as *autonomous* attributes. Autonomous attributes, such as position, vary over time and must be updated implicitly before their values are consumed for subsequent processing. An important characteristic of autonomous attributes is that they should not need to be set explicitly by simulation objects modeling real-world entities. They should instead be updated transparently by the simulation environment as time advances.

It is important for a number of reasons that autonomous attributes be current. First, from a software engineering point of view, the data and knowledge associated with a simulation entity should be current at any point in time during the simulation. Clearly, an object's location is a piece of data that must be kept updated. For instance, if a symbolic snapshot or dump of the simulation is produced for simulation time t, it is imperative that the databases of all simulation objects reflect accurate information for time t. This concern becomes more apparent when a graphics display is generated. When the graphics display is updated, it must present accurate and current data for all simulation objects. Therefore, it is necessary that the locations of all displayed objects be current. If their positions do not correspond to the displayed time, then the locations must be updated before the objects are

displayed. A third critical consideration pertains to other uses of an object's database. In a given simulation it is likely that many computations depend on an object's location. Whenever any autonomous data is consumed by other objects, that data must be current.

The considerations described above hold for all autonomous attributes. The underlying requirement is to ensure one of two situations: (1) whenever the simulation clock is advanced, all time-dependent data is updated, or (2) whenever time-dependent data is accessed, it is updated if it is not current. In theory, these requirements are stated easily; however, in practice, achieving either goal burdens the development of the simulation. Programmers are required to be aware of all autonomous attributes and procedures for maintaining these attributes. Depending on which of the above maintenance strategies is adhered to, the simulation developer must procedurally invoke the appropriate update routines whenever the clock is advanced or an autonomous attribute is retrieved.

Through our experiences in developing object-oriented simulation models, we have concluded that these bookkeeping and maintenance tasks are rarely performed diligently. As the size of the simulation system grows, it becomes increasingly difficult to manage these artifactual activities. Failure to provide consistent maintenance results in both blatant and subtle imperfections in the simulated world. Inconsistencies are most obvious when a graphics display is plotting the progression of the simulation. A fleet of vehicles that should be adhering to a given formation is sometimes displayed with one or more of the vehicles out of synchrony with the rest of the fleet. This behavior usually indicates that some, but not all, of the moving vehicles were updated. A less obvious manifestation of an inconsistency occurs when the graphics display shows position information that does not coincide with the simulation's database, although the time of the graphics frame matches the simulation clock. Inconsistencies can also occur strictly within the simulation processing if time has advanced without updating an autonomous attribute accessed for a subsequent computation.

One goal of this research has been to try to eliminate the programming tasks of updating autonomous attributes and invoking artifactual procedures. In Section 4.1 we discuss our approach to this problem: the declarative specification of autonomous attributes supporting transparent *update-on-demand*.

3.2. Simulation of Asynchronous Activities

In this section we characterize another deficiency of object-oriented simulation languages related to the scheduling of future events. Many activities and events in a simulation are triggered by an object's local actions. For greater fidelity, a simulated object should schedule *only* those activities that the analogous real-world object can control. This

convention promotes an isomorphic relationship between the simulation and the world the simulation is modeling. There are, however, other physical phenomena taking place in the real world that require simulation and monitoring.

Consider the task of modeling an airborne vehicle that detects other objects whenever they are within the aircraft's sensing range. An intuitive approach for modeling the concept of *sensing* would represent the aircraft and other entities as simulation objects. The aircraft are assigned a sensing range and position. We write a behavior to compute the distance between the aircraft and all other objects to determine which (if any) are within sensing range. However, we are faced with the problem of deciding when to invoke the sensing behavior and which object should call it. In the real world, an aircraft doesn't issue explicit commands to compute its distance to other objects; rather, it detects objects that are within its range as a *side effect* of flying its course and moving within sensing range. Therefore, it is unnatural for the aircraft to invoke the sensing behavior. It is even less appropriate for the detected objects to invoke sensing because sensing is an action of the aircraft, not of the detected objects.

A similar problem occurs when attempting to determine whether an aircraft has collided with another object or run out of fuel. In the real world, a collision between objects and the consumption of fuel happen as side effects of the aircraft's movements. Events that take place as side effects of autonomous activities do not fit naturally into the existing event-based simulation framework.

We can force side effects, or *non-intentional* events, to take place by writing a control loop that is executed whenever simulation time advances. In the control loop we check for conditions that trigger side effects and call the appropriate behaviors when the proper conditions exist. For example, at each increment of the simulation clock we compute the distances between an aircraft and all other simulation objects to determine which objects are within the aircraft's sensing range at that particular time. We also compute the aircraft's fuel level to see if it has run out of gas in midair, and we calculate its distance to all other objects to learn whether it has collided. However, this approach has several drawbacks. First, it is unnatural to explicitly compute an aircraft's fuel level from many different places in the simulation code. Second, it is inefficient to perform these calculations at *every* simulation time update. Finally and most importantly, if we perform these computations *only* when the clock is advanced, we may corrupt the integrity of the simulation. It is possible, for example, for a collision between two moving objects to go undetected if the distance between the two objects is not computed at precisely the right time. This situation occurs if one object is within sensing range of another for some small time interval, which

begins after one clock update and ends before the next update. For all of these reasons, the SWIRL simulation system (Klahr et al., 1982b), introduced an auxiliary object called the *scheduler*.

In SWIRL, the responsibility of invoking side effects, or non-intentional events, was assigned to the scheduler. A scheduler object preplanned and scheduled events based on initial attribute values of simulation objects. This approach removed from the simulation object the burden of checking for certain conditions, and instead placed it on an auxiliary object that has no real-world analogy. This strategy was conceptually cleaner because it removed from real-world simulation objects bookkeeping tasks which have no real-world correlate. However, it was not an ideal solution because it still did not guarantee that side effects would take place at exactly the right time. If, for example, the scheduler planned, at some future time, a collision time between two moving objects based on the initial velocities of those objects, the collision would take place at the planned time whether or not their velocities changed during execution of the simulation. Clearly this leads to incorrect simulations. Although most processing in a Ross simulation is event-driven, programmers have been limited to modeling non-intentional activities as *time-driven* events; i.e., fixed or variable time increments have triggered the evaluation of conditions, rather than the fulfillment of a condition triggering a subsequent action.

To overcome the deficiencies of the scheduler approach, it is necessary to monitor autonomous activities so that when certain conditions or constraints are fulfilled, specific actions are scheduled or executed. We refer to this monitoring process as an *asynchronous* activity. The difficulty involved with this activity also stems from the continuous aspect of time, which is not captured in discrete event simulation. Examples of asynchronous activities introduced above include the consumption of fuel by a moving object; the collision of two moving objects; and the sensing of one object by another. In each of these situations, the simulated objects must be alerted when they run out of fuel, a collision occurs, or one object is within the sensing range of another object.

For a programmer to maintain the scheduling of these asynchronous activities (with or without a *scheduler*) a great deal of programming time and effort is required. Once the asynchronous activities and dependent attributes are identified, the programmer must be aware of the processes being scheduled and of existing dependencies. Furthermore, the programmer must manage the scheduling and unscheduling of triggered events. As in the case of maintaining autonomous attributes, system designers and implementors become inundated with details of artifactual simulation processing that do not contribute to the primary simulation effort, yet must still be accounted for. These issues and dependencies must be considered; otherwise the simulation and its results will be worthless.

A second aspect of this work has been developing *demon* processes for managing asynchronous activities in simulations. In Section 4.2 we present the operational aspects of this facility, which allow a simulation developer to specify asynchronous activities. The demon facility adopts a declarative approach for specifying condition fulfillment, resulting actions, attribute dependencies, optimization considerations, termination conditions; and for automating invocation of updating procedures. Once set into motion, the demon facility transparently schedules and reschedules monitored asynchronous events.

3.3. Interfacing simulation and graphics

The artifactual simulation activities we have identified are necessary for maintaining consistency *within* the state of the simulation. We now consider the consistency between the simulation and its graphics presentation.

A simulation should display its simulated world with appropriate detail and continuity to allow visual comprehension of its behavior. It is tempting to think of the display as a window into the state of the simulation, whereby whenever anything is changed by the simulation, it is immediately visible on the display. Unfortunately, producing this illusion in a sequential object-oriented simulation system requires considerable effort. We have analyzed two traditional approaches for keeping a display up-to-date with respect to the simulation: (1) the *display processor* approach, and (2) the *incremental graphics* approach. In this section, we characterize each approach, and identify its strengths and weaknesses. We conclude this section by introducing a new strategy that we developed that provides the benefits of both approaches.

The *display processor* approach views graphics display as an independent process that redisplays the entire state of the simulation for each new graphics update. The simulation runs at its own pace with no awareness of any graphics output. The simulation maintains the state of various *graphic attributes* of simulation objects, which are retrieved by the display processor when generating a new graphics frame. The attraction of this approach is that the simulation need not concern itself with producing graphics; it simply executes and keeps its state current. The display processor keeps the simulated world current by redisplaying its entire state at regular *graphic update* intervals. However, the attributes must be in a consistent state when accessed. This requirement necessitates synchronization between the model and the display processor. The frequency of frames, that is, of graphic updating, is essentially determined by the display processor. The simulation, however, must also have some control over when new frames occur, both to ensure consistency and to allow explicit control over the interval between new graphics frames.

The main disadvantage of this approach is that every new frame requires redisplaying the entire simulation state. Although previous Ross simulations have taken this approach, its efficiency is directly related to the dynamics of the graphics display. If only a few of the displayed objects have changing *graphic attributes*, then resources are wasted by redisplaying static features. Furthermore, this strategy is not compatible with many current graphics software standards, such as Core (Newman and Dam, 1978) and GKS (ANSI, 1985). These graphics standards are based on persistent data structures called *segments,* which are mapped to conceptual objects in the simulation. Graphic segments supply the display characteristics necessary to generate an image of the corresponding simulation object. Rebuilding new segments for each graphics frame contradicts the design of persistent segments, which instead should have their segment attributes modified. Nevertheless, for those objects whose graphics representation changes drastically over time, new segments must be constructed.

In the display processor approach, the graphics processing is decoupled, either physically or conceptually, from the simulation. The display processor simply queries the simulation for necessary data. We contrast this method with the *incremental graphics* approach. In this strategy, the simulation model generates graphics output as it executes. The simulation, therefore, controls when and how changes are made to the graphics image. Because the simulator knows when it is necessary to modify the graphics image, this approach results in greater efficiency, which in turn may improve the appearance of the display. For example, if a given graphic attribute is not affected between frames, it will not be redisplayed. This strategy reduces redundant updating, which in turn reduces both graphics processing and potential visual distraction. It may also result in better graphics dynamics; when an event in the simulation causes graphics output, the simulation can use special graphic techniques to highlight the meaning of the event for the user. These techniques are more difficult to accomplish with the display processor approach because the semantics of an event are usually lost by the time the display processor produces its next graphic update. The disadvantage of incremental graphics is that the model must perform its graphics output explicitly. Designers and programmers of the simulation must be aware of the integrated graphics processing and must contend with the decisions concerning graphics display, such as what simulation changes should affect the graphics image and how those changes should be manifested graphically. Therefore, strict graphics conventions and standards must be designed and adhered to when adopting an incremental graphics approach.

In Section 4.3, we present our alternative to the display processor and incremental graphics approaches described above. Our *graphics-delta* approach combines the merits of the other two by providing automatic facilities for maintaining the display history of graphics attributes. The graphics-delta approach allows a user to declaratively specify simulation objects and attributes, and define corresponding graphics images that support the simulation's graphic display.

In this section we have presented three problem situations facing the developer of object-oriented simulation systems: (1) maintenance of autonomous attributes, (2) simulation of asynchronous activities, and (3) interfacing the simulation model with its graphics presentation. In the next section we present the methodology and techniques we have developed to help reduce the amount of artifactual processing that simulation programmers must perform to address these issues.

4. DECLARATIVE FACILITIES FOR ARTIFACTUAL PROCESSING

The results of this work are object-oriented simulation facilities to automate artifactual programming tasks. As we will discuss in Section 6, these facilities were developed as extensions to the Ross language. Although the artifactual tasks we have described are necessary, our objective is to make this type of processing as transparent to the designer, programmer, and user as possible. Toward this end, our methodology supports a *declarative* rather than *procedural* approach to maintaining object-oriented simulations (Hill and Roberts, 1987). The artifactual programming activities we have discussed are based on procedural routines. That is, the person producing the simulation code must consider the consistency issues we have raised throughout the entire software development and implementation stages, and provide procedural methods for maintaining consistency. The declarative approach allows the simulation developer to declare those attributes, behaviors, dependencies, and asynchronous conditions that must be managed. Our simulation facilities automatically perform the proper updating of autonomous attributes, monitoring of asynchronous conditions, and display of new consistent graphics frames. Below we describe the declarative mechanisms facilitating these automatic processes.

4.1. Update-On-Demand

We have developed *update-on-demand* facilities to help automate dependency management. Update-on-demand assures that whenever an autonomous attribute is referenced, the attribute is automatically and transparently made current before a value is returned. This approach ensures consistency of all time-dependent attributes throughout the simulation. It also eliminates artifactual code that must otherwise perform explicit attribute updating.

To enable update-on-demand processing, the simulation developer must perform the following: (1) declare the autonomous attributes of each object, and (2) supply a procedural behavior dictating how the updated value of an autonomous attribute is computed. This behavior, however, is never invoked explicitly in the simulation code. The procedural code is automatically invoked when the autonomous attribute is referenced. Therefore, the implementor need not be concerned that an autonomous attribute may need updating.

We have optimized our update-on-demand facility to perform the necessary update only when the value of the autonomous attribute is out-of-date, thereby minimizing the computational overhead if the attribute value is already current. In Section 5, we demonstrate the use of the update-on-demand facility by providing a comprehensive example comparing simulation code with and without automatic updating. Because the example described in Section 5 explains the use of all three of our declarative Ross extension packages, we postpone our detailed discussion of the example until we have described the remaining extensions in subsections 4.2 and 4.3.

4.1.1. Identifying Second-Order Dependencies on Time

New values for autonomous attributes generally depend on time as well as the values of other attributes. If an autonomous attribute is a function of only the instantaneous values of other attributes, there is no problem: recomputing the autonomous attribute (on demand) will always produce the right value by using the current values of the attributes on which it depends. If, however, an autonomous attribute depends on the *history* of past values of some other attribute, then a second-order dependency exists. For example, position depends on previous values of speed and, in particular, on when speed was last changed. That is, position can be thought of as a function of initial position and the sequence of previous values of speed along with the times at which speed changed.

Attributes on whose history other attributes depend are referred to as *history-affecting attributes*. Our work addresses those history dependencies that occur among attributes of the same object. Such intra-object dependencies are common and important enough to warrant special treatment. Furthermore, automatically updating these localized dependencies can be viewed as a way of increasing the encapsulation effect of objects. Non-local (inter-object) dependencies can still be handled by explicit update, which is how most simulation environments handle *all* dependencies. We discuss this assumption further in Section 7. Note that those *history-dependent* attributes that depend on past values of history-affecting attributes are always autonomous attributes. Because non-autonomous attributes are set explicitly by the simulation code, the dependencies considered here do not apply to them.

Normally, changing a history-affecting attribute should not cause its history-dependent position to be updated immediately. Only at a later time, when the object has moved using the new value of speed, should position reflect the new speed. However, if an object's position is out-of-date when its speed is changed and its position subsequently gets

updated within the same simulation time, then the new position will erroneously reflect the new speed even though that speed has been in force for zero time. If, on the other hand, the position has already been updated at a given simulation time before the speed is changed, then the new position will correctly reflect the old speed. The simulation must therefore update position (whether or not it is about to be retrieved) before updating speed to prevent speed from affecting position until time has advanced.

A history-affecting attribute can be changed several times within a single simulation time without updating its history-dependents more than once. This is because such dependencies are always functions of time; a history-affecting attribute only affects its history-dependents when time has elapsed. Multiple settings of a history-affecting attribute, such as speed, within a single simulation time are instantaneous events and have no effect. Only the final value counts and only after time has elapsed. However, to behave properly, history-dependent autonomous attributes must be updated *before* the first update of one of their history-affecting attributes at a given simulation time.

4.1.2. Maintaining Second-Order Dependencies

The above problem can be thought of as a limited form of constraint propagation (Borning, 1979; Stefik et al., 1986). In the example discussed above, every time speed is updated, position should be updated first. One solution for maintaining history-dependent attributes is to represent pairs of dependent attributes such as speed and position, so that whenever speed is modified, the value of position is updated first. This approach represents dependencies explicitly; its disadvantages are that these lists must be maintained as the simulation code evolves, and implicit updating loops may inadvertently be created.

We have noted, however, that only autonomous attributes require automatic update as a result of second-order effects. Therefore, second-order inconsistencies can be avoided by automatically updating *all* autonomous attributes before updating any attribute on whose history an autonomous attribute may depend. To prevent the undesirable effects of second-order dependencies, the simulation developer must (1) declare autonomous attributes (as described in Section 4.1) and (2) declare history-affecting attributes during object specification. During simulation processing, whenever a history-affecting attribute is modified, the object's autonomous attributes are automatically updated using the supplied update behaviors. This approach represents attribute dependencies implicitly, and therefore does not require keeping lists of such dependencies up to date as the simulation code evolves. Using this approach, some extra overhead may be incurred because not *all* autonomous attributes depend on the history of *all* history-affecting attributes. However, the

overhead is bounded by eliminating inter-object dependencies. The examples in Section 5 will provide a concrete demonstration of update-on-demand for autonomous attributes and the transparent interaction triggered by updating history-affecting attributes.

4.2. Demon Processes

In discrete event simulations, events can be scheduled to take place at specific times. For example, an aircraft following a fixed track plan can be scheduled to change its speed, direction, and altitude at precomputed times. These *intentional* events can in turn trigger other events, but they cannot conveniently trigger *non-intentional* events, or *side effects*. In the real world, side effects do not occur at specific preplanned *times*, or as the direct consequence of other *events* taking place; they occur if and when certain *conditions* arise. For this reason, our demon facility allows the simulation programmer to define side effects in terms of situation-action rules. In this section, we describe the operational aspects of our demon facility, which enables the simulation programmer to specify non-intentional asynchronous events in a natural and straightforward manner.

A demon is a process that surveys the simulation state: when a particular situation arises, the demon wakes up, takes action, and then goes back to sleep. We have implemented demons as behaviors that reschedule themselves automatically and transparently throughout execution of the simulation.

To define a demon, the simulation programmer must provide the following: (1) a situation, or *action-triggering condition*, to watch for; (2) an *action* to take; (3) which *class* of objects is directly involved and, optionally, which class is indirectly involved; and (4) which attributes of the objects involved have an effect on the behavior of the demon (these are referred to as *demon-affecting attributes*). The programmer should provide two additional specifications whenever possible: (5) a *continuation condition*, which must be satisfied if there is any chance the action-triggering condition may exist at any future time; and (6) a *prediction function*, which provides an estimation of how soon the action-triggering condition will come into existence.

The action-triggering condition and the action must both be supplied as behaviors. They combine to form the situation-action rule describing the side effect. The continuation condition, also supplied as a behavior, determines whether the action-triggering condition can occur in the future. If this condition is not satisfied, the prediction function will not be used to estimate when the action-triggering condition will come into existence because, given the current state of the simulation, it will never exist.

The prediction function provides an estimation of how far into the future the situation will exist. In the worst case, this function computes the minimal amount of time until the situation can possibly come into existence. The prediction function should be supplied for two important reasons. First, without this function, the action-triggering condition would have to be checked at fixed time intervals. This function should provide a better prediction of when the situation will occur, and therefore, optimize the surveillance process. Second, and more importantly, this function can guarantee that the action-triggering condition will not go undetected. By computing successive approximations to the time at which the action-triggering condition will come into existence (in the worst case), this function can ensure that the time interval in which the situation exists will not be missed. It therefore enables the continuous aspect of time to be captured in the discrete simulation.

The demon-affecting attributes list (above) must contain the names of all attributes whose values are used in the following: the action-triggering condition behavior, the prediction function, and the continuation condition. The prediction function and continuation condition predict when and if the action-triggering condition will exist, based on the current state of the simulation at the time for which they are invoked. It is possible that they may not expect the action-triggering condition to arise in the future; however, conditions may subsequently change, creating a new possibility that it will exist. (For example, an aircraft that lands will not run out of fuel and crash. However, if it subsequently takes off again, there is a new possibility that it will eventually run out of fuel.) Therefore, all demon-affecting attributes are monitored. Whenever the value of any of these attributes changes, the action-triggering condition is checked for immediately. If it does not exist, the continuation condition and prediction function are called to provide a revised estimation of when the action-triggering condition will come into existence, based on the new attribute values.

The aircraft's speed and rate of fuel consumption are the attributes that affect the precise time at which the vehicle will run out of fuel. Therefore, they must be declared to be demon-affecting attributes. The action-triggering condition is defined as a fuel level of zero while the aircraft is in midair, and the resulting aircraft action is to crash. Additionally, an estimation function should be supplied that computes how soon the aircraft will deplete its fuel supply, based on current fuel level, speed, and rate of fuel consumption; and a continuation condition should check whether the aircraft is still flying.

Our demon facility offers several advantages over the traditional methods discussed in Section 3.2. First, no artificial auxiliary objects are created. In this way, the demon facility promotes a modeling methodology that contains an isomorphic relationship between the simulation and the world it represents. Thus, function calls invoking artifactual code do not appear in the simulation. Second, demon definitions are self-contained. This encourages modular programming practices, making it easy to add and remove demon definitions without having to modify the rest of the simulation code. Third, the user-supplied prediction function will often optimize the surveillance process without sacrificing accuracy; the action-triggering condition can be evaluated less frequently without running the risk of missing detection of that condition altogether. Fourth, and most importantly, varying the increment size of the simulation clock no longer affects the simulation outcome.

4.3. Delta Approach Interfacing Graphics and Object-Oriented Simulations

In Section 3.3 we discussed two methods for graphically displaying the execution of an object-oriented simulation. In the display processor approach, the simulation runs without performing explicit updates while the display processor continually redisplays the entire state of the model. The second alternative, incremental graphics, forces the model to update the display explicitly whenever any graphics-related event occurs. In this section, we describe a third alternative, which we call the *graphics-delta* approach. Graphics-delta is a compromise between the other two extremes. The simulation performs no explicit graphics processing; instead, (1) the conceptual display processor determines what graphical changes are necessary to produce a new graphics frame, and (2) incremental graphic updates are generated as needed. The operation of our graphics-delta approach is detailed below and an example is provided.

In the graphics-delta approach, the simulation performs no explicit graphics output, but simply updates attributes. During simulation development, the programmer declares *frame-triggering attributes*, that is, those attributes that affect the display. Identifying an attribute as a frame-triggering attribute indicates two things to the simulator: (1) that the value of those frame-triggering attributes will be mapped to appropriate display attributes for generating a graphics display, and (2) whenever the value of one of those frame-triggering attributes changes, a new graphics frame will be displayed. Therefore, declaration of frame-triggering attributes implicitly identifies graphically significant events, and changing the value of a frame-triggering attribute transparently invokes the display processor. Frame-triggering attributes keep a record of their value at the time they were last displayed.

In this way the display processor, rather than redisplaying the entire simulated world, instead uses the history of the simulation's frame-triggering attributes to construct a graphics-delta between the current graphic state of the simulation and the last displayed frame. This graphics-delta is used to apply incremental graphic updates, thereby retaining the efficiency of the incremental approach.

The recording of history for frame-triggering attributes makes this approach efficient. Further, it reduces redundant updating when a frame-triggering attribute is set but the value has not changed. Only attributes whose values differ from frame to frame will cause graphic updates. The graphics-delta approach retains the conceptual simplicity of the display processor approach because each displayable object is simply asked to redisplay itself; however, this redisplay retains the efficiency of the incremental approach due to the maintenance of frame-triggering attribute history. Synchronization is afforded by explicitly invoking the display processor whenever a graphically significant event occurs. These events are recognized by the simulator's event scheduling mechanism and force the model into a consistent state before each graphic update.

From the programmer's point of view, the only necessary tasks are to declare the frame-triggering attributes of an object and define procedural attributes which are called *primary-image* and *secondary-image*. These attributes, when evaluated, call hardware-dependent graphics routines to display an icon corresponding to the simulation object. The primary-image procedure displays the default image of an object, such as a radar icon. If an augmented or alternate image is desired, then a secondary-image is used—for instance, to display a sensing range or sensing envelope for a cluster of radar. These extensions automatically maintain the history of the frame-triggering attributes, schedule a new graphics frame when a frame-triggering attribute is modified, and transparently evaluate a frame-triggering object's primary and secondary images, thereby displaying a new graphics frame. Examples of the use of frame-triggering attributes and automatic frame generation are presented in the next section.

5. A SAMPLE SCENARIO DEMONSTRATING ROSS SIMULATION EXTENSIONS

In this section we present a small simulation implemented in the Ross language. With this example, we hope first to give the reader a sense of how declarative Ross extensions are used within a simulation system and of the functionality provided by these capabilities. Second, we want to contrast the traditional procedural approach for maintaining consistency with the declarative methodology we are advocating. For these reasons, we present a simulation employing procedural methods first, followed by a series of subsections showing how capabilities provided by each of the three Ross extension packages are integrated into the scenario.

In the example, two Remotely Piloted Vehicles (RPVs) fly through hostile territory that contains two enemy radars. The RPVs traverse the airspace on a fixed trajectory. Upon sensing an enemy radar, an RPV sends a warning message to its partner and upon receipt of the warning communication, the partner increases its speed. Each RPV consumes fuel as it flies, and it crashes if it runs out of fuel in midair.

The class and instance hierarchy for this scenario appears in Figure 5.1. The classes *something* and *nclock* are built into the Ross system in a manner similar to the *vanilla* flavor in the Flavors package. The bold-type objects are the classes and instances specific to the simulation. The bold-face leaves of the tree represent instances. The simulation contains two instances representing RPVs (Rpv1 and Rpv2), and two radar instances (Radar1 and Radar2). Rpv1 and Rpv2 are members of the class *RPV*, which in turn is a subclass of the class *Moving-object*. Radar1 and Radar2 are members of the class *Radar*, which is a subclass of *Fixed-object*. Warning messages are represented as temporary objects created as instances of the *Communication* class. Each instance of the RPV class has the following attributes: *position, speed, trajectory, velocity, mpg, fuel-level, flight-status*, and *sensed-defenses*. The *mpg* attribute specifies the rate of fuel consumption, and *sensed-defenses* is a list of all objects the RPV is currently sensing. *Flight-status* indicates whether an RPV is inflight, refueling, or crashed.

In Figure 5.2, we show the values of the attributes of Rpv2 throughout the course of a simulation executed from time 0 to time 15. The dashes indicate that those attribute values were not computed for the corresponding times. Notice an additional attribute, *plans*, a Ross default attribute, which maintains a list of events that are scheduled to be executed by the corresponding object.

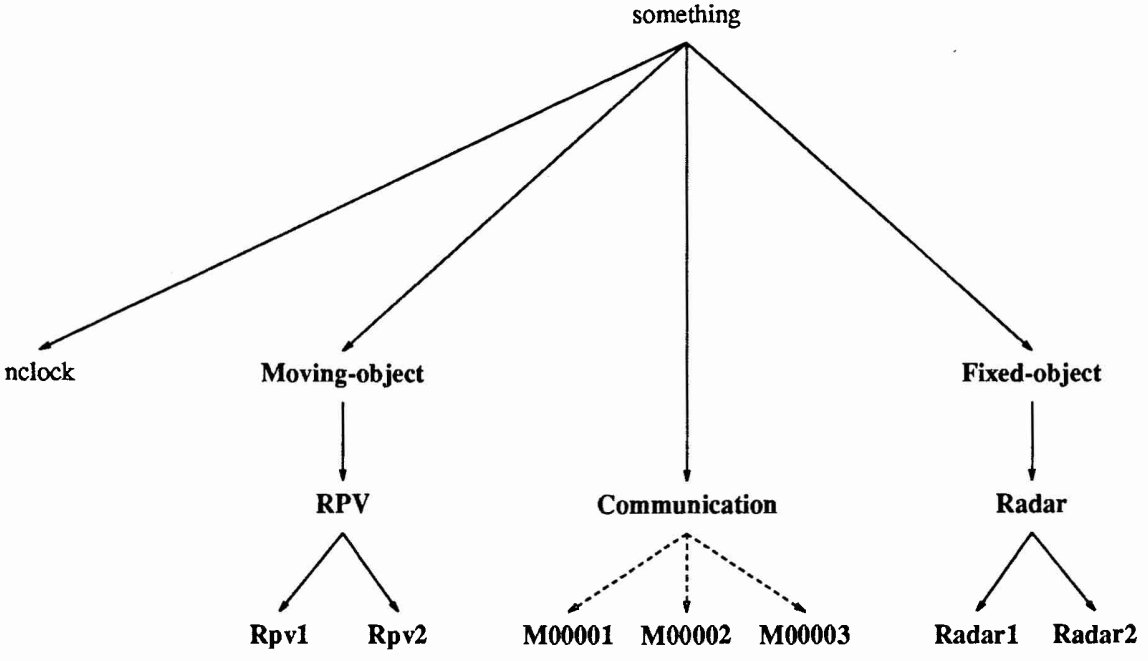

Fig. 5.1—Class and instance hierarchy

$stime	trajectory	speed	mpg	velocity	position	fuel-level	sensed-defenses	flight-status	plans
0	(1.0 1.0)	5.0	10.0	(3.5 3.5)	(0.0 3.0)	7.0	()	in-flight	()
1	-	-	-	(3.5 3.5)	(3.5 6.5)	6.5	(radar1)	-	((2 (react to sensed radar1)))
2	-	-	-	(3.5 3.5)	(7.0 10.0)	6.0	-	-	((3 (send warning communication)))
3	-	-	-	(3.5 3.5)	(10.6 13.6)	5.5	()	-	()
4	-	-	-	(3.5 3.5)	(14.1 17.1)	5.0	(radar2)	-	((5 (react to sensed radar2)))
5	-	-	-	(3.5 3.5)	(17.6 20.6)	4.5	()	-	((6 (receive communication m00002)) (6 (send warning communication)))
6	-	6.0	-	(4.2 4.2)	(21.2 24.2)	3.9	-	-	()
7	-	-	-	(4.2 4.2)	(25.4 28.4)	3.3	-	-	-
8	-	-	-	(4.2 4.2)	(29.6 32.6)	2.7	-	-	-
9	-	-	-	(4.2 4.2)	(33.9 36.9)	2.1	-	-	-
10	-	-	-	(4.2 4.2)	(38.1 41.1)	1.5	-	-	-
11	-	-	-	(4.2 4.2)	(42.4 45.4)	0.8	-	-	-
12	-	-	-	(4.2 4.2)	(46.6 49.6)	0.2	-	-	-
13	-	0.0	-	(4.2 4.2)	(50.9 53.9)	0.0	-	crashed	-
14	-	-	-	(0.0 0.0)	(55.1 58.1)	0.0	-	-	-
15	-	-	-	(0.0 0.0)	(55.1 58.1)	0.0	-	-	-

Fig. 5.2—History of attribute value changes for Rpv2

Figure 5.3 contains a representation of the simulation at simulation times 0, 1, 3, and 9. Time 0 is the initial configuration. At times 1, 3, and 9, the significant events listed below occur in the simulation. These events (and others) are reflected in the attribute values of Rpv2 shown in Figure 5.2.

- At time=1, Rpv2 detects Radar1 and continues sensing it until time=3.
- As a result of the sensing, Rpv2 sends a warning communication to Rpv1, which is received at time=3. Rpv1 subsequently increases its speed.
- At time=9, Rpv1 runs out of fuel and crashes.

The simulation code contains the following control loop, which invokes time-driven events (the code for this entire scenario is contained in Appendix B):

```
(tell nclock when receiving (tick and update >n times)
      (loop for i from 1 to n
            do (tell !myself tick and update)))

(tell nclock when receiving (tick and update)
      (tell !myself tick)
      (loop for vehicle in (ask rpv recall your offspring)
            do (tell !vehicle update your fuel-level)
               (tell !vehicle update your position)
               (tell !vehicle update your velocity)
               (tell !vehicle check flight status)
               (tell !vehicle sense)))
```

The nested iterative loop "tick and update" sends several Ross messages to each RPV to tell it to update its time-dependent attributes (fuel-level, position, velocity), and then to check whether its flight status has changed (i.e., whether it has run out of fuel and crashed). Finally, an RPV determines which objects it is currently sensing by computing its distance to every radar object.

There are several drawbacks and problems with implementing the simulation using such a control loop. First, it is unnatural to send messages to RPV objects telling them to update their attributes. Explicit updating of time-dependent attributes has no direct correlate in the real world, and therefore is unnatural in the simulation code. Second, the value of each time-dependent attribute is recomputed at every tick of the simulation regardless of

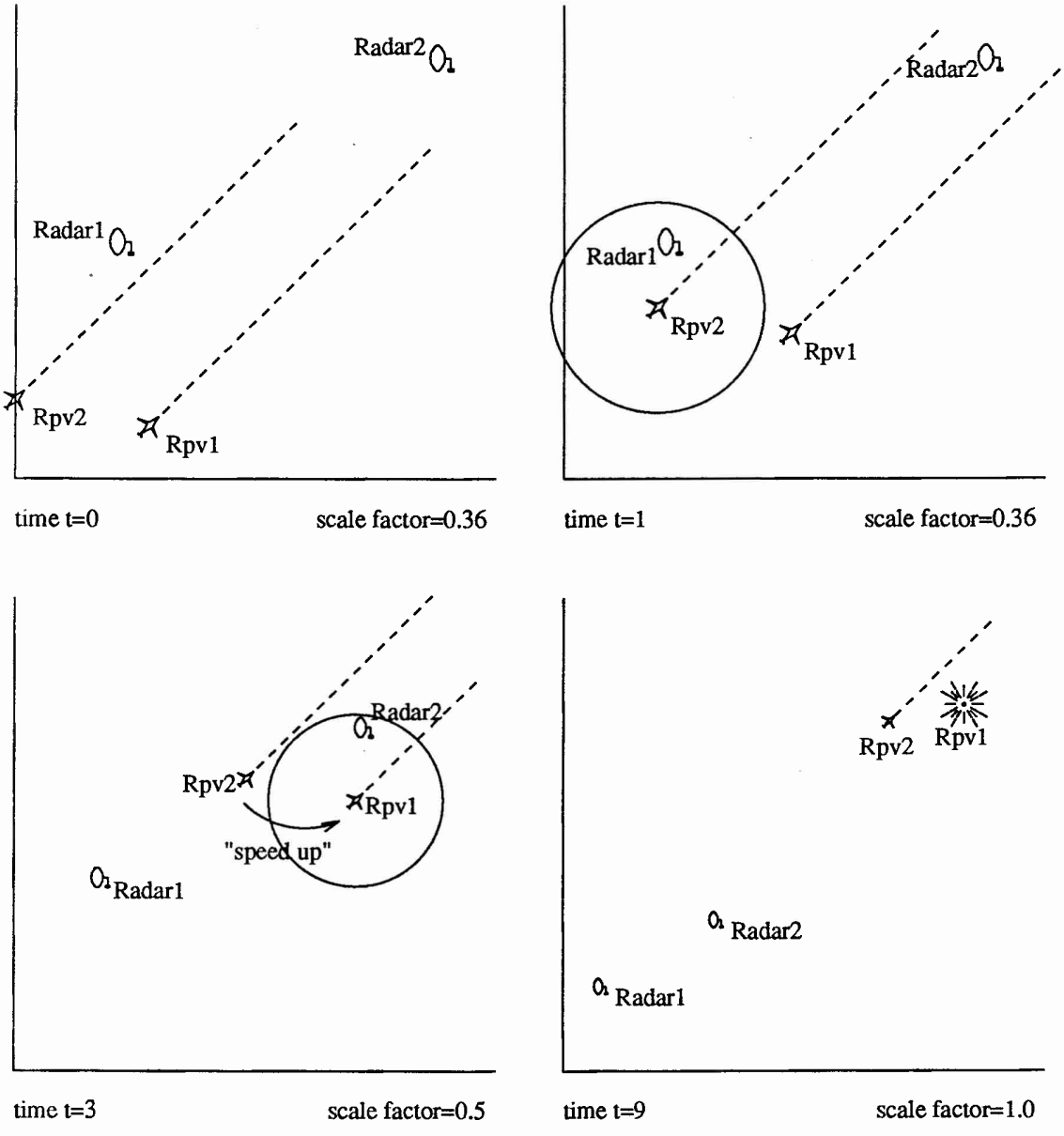

Fig. 5.3—Representation of the simulation at times 0, 1, 3, and 9

whether or not its value is consumed by another computation. Frequently, this duplicated processing is wasteful. For example, a moving vehicle's fuel level does not need to be recomputed at every tick if it is only used to indicate when the vehicle has run out of fuel. Third, even though we have been careful in ordering the computations for time-dependent attributes, these values will still not always be computed in exactly the correct order, and therefore the simulation will not be completely accurate. We have placed the velocity computation after the position computation because if the velocity of an object changes at a certain time, its position for that same time must be computed using the previous velocity — the new velocity hasn't actually taken effect yet. Similarly, we need to make sure that fuel level is always computed before speed is changed because fuel level depends directly on speed. However, Ross will not allow this processing sequence without including artifactual code to check and update values. When the simulation clock is advanced, Ross executes the *plans* for each object before executing the control loop. Thus, if an RPV is scheduled to "receive a warning message" at time *t*, and reacts to the message by increasing its speed, it will proceed to change its speed before its fuel level has been computed using the previous value of speed. Careful examination of the fuel levels computed for Rpv2 in Figure 5.2 (beginning at time 6) shows that they are incorrect. Without a way to specify that certain computations must be performed before scheduled events are processed, it is not possible to build a completely accurate simulation. In addition, each RPV's position is updated one time too many. The position should not change after an RPV has crashed. Finally, varying the ticksize (or time increment) of the simulation will further distort the outcomes because computations depend directly on advancing the simulation clock.

In the next three subsections, we modify the code for this simulation to incorporate our Ross extensions. In Section 5.1, we add update-on-demand processing to eliminate explicit updating of each autonomous and history-affecting attribute. Section 5.2 shows demons defined to initiate and terminate sensing, and to monitor fuel consumption. We demonstrate that a control loop is not needed when automatic updating and demons are used properly. In addition, we show that varying the ticksize of the simulation no longer affects the simulation's outcome. Finally, in Section 5.3, we illustrate how graphics frames can be generated automatically, either for each update of the simulation clock or for user-declared *graphically significant* events.

5.1. Update-on-Demand Processing

The simulation programmer cannot always control the order in which computations are performed in a simulation. It is not possible to guarantee that they will be carried out in the correct order. However, by including update-on-demand extensions and by specifying attribute dependencies, we can ensure that computations will always be performed in the correct order. In addition, the need to explicitly update each time-dependent attribute at each update of the simulation clock is eliminated.

To include update-on-demand processing in our simulation, we must first classify all *autonomous* and *history-affecting* attributes. Autonomous attributes are those whose values vary directly or indirectly as a consequence of the passage of time. History-affecting attributes are those whose history affects the values of autonomous attributes.

In our simulation, RPVs are the only active objects, so we need consider only their attributes. Because RPVs are moving objects, their positions change as time advances. They also consume fuel as they move; therefore, their fuel levels depend on the passage of time. Although velocity is not directly dependent on time, velocity is regarded as an attribute that stores an intermediate computation subsequently used for computing position. Therefore, velocity must be categorized in the same way as position, namely, as an autonomous attribute. Because position, velocity, and fuel-level are classified as autonomous attributes, we must also supply "update" functions to bring these values up to date.

Next we must determine which attributes are history-affecting. Position is defined in terms of velocity, another autonomous attribute. However, velocity is defined in terms of speed and trajectory; therefore, speed and trajectory are history-affecting and their values are only changed when the simulation code modifies them explicitly. The other autonomous attribute, fuel-level, is defined in terms of speed and mpg. Therefore, mpg is also a history-affecting attribute. (We assume that mpg does not depend on speed.)

It is helpful to map out the relationships between various attributes by drawing a value-dependency graph. Figure 5.4 depicts the dependencies between the attributes we have just examined. This approach forces programmers to consider the intended semantics of the simulation early in the development effort. Even without automatic updating facilities, these considerations must be resolved. If the characteristics and behaviors of the simulation objects are understood as part of the system design, many conceptual processing errors can be avoided.

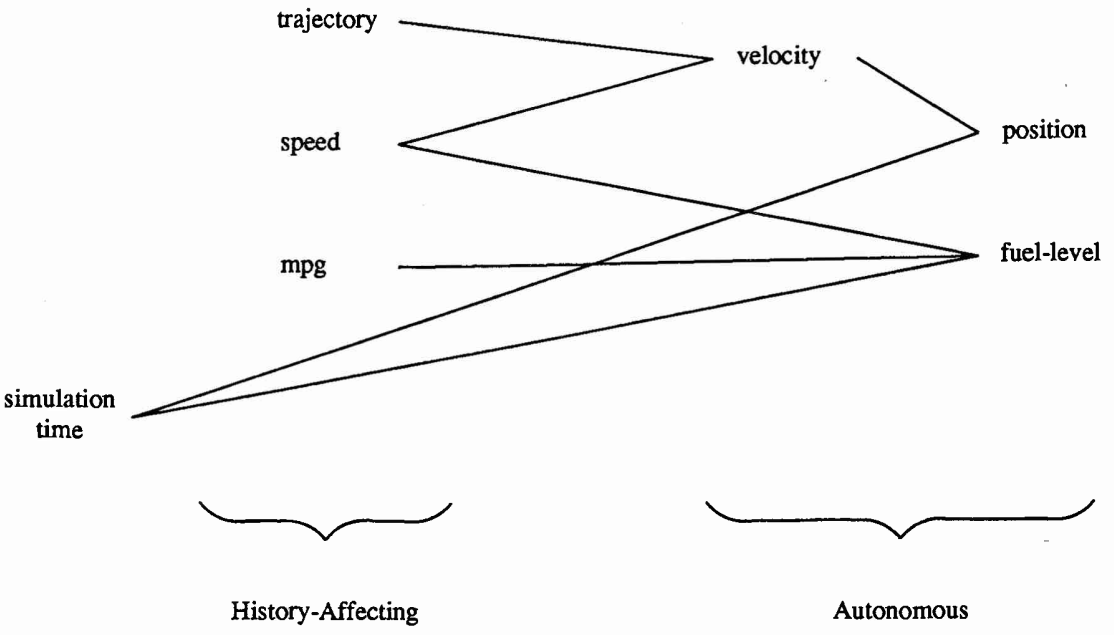

Fig. 5.4—Attribute dependency graph

To use update-on-demand processing in a simulation, a programmer must follow the steps given in Section 1 of Appendix A. For our example, the necessary additions are given below. The step numbers correspond to the steps described in Section 1 of Appendix A. Note, however, that these are not *all* the declarations necessary for producing our example simulation; rather, they are the ones required for augmenting an existing simulation with update-on-demand processing. The "core" set of class declarations (without any extensions) can be found in the "objects.l" file of Appendix B.1.

(1) Create all top-level classes as subclasses of the class **simulator**. For example,

```
(tell simulator create new generic moving-object)
(tell simulator create new generic fixed-object)
(tell simulator create new generic communication)
```

(2) Create the RPV class as a subclass of the **monitored-object** class, because its instances have autonomous and history-affecting attributes.

```
(tell monitored-object create generic rpv)
```

(3) Specify which attributes of RPVs are autonomous or history-affecting by placing them on the **$monitored-attributes** list for RPVs:

```
(tell rpv set your $monitored-attributes
      to (fuel-level mpg position speed trajectory velocity))
```

(4) Place the names of all autonomous attributes on the **$autonomous-attributes** list for RPVs:

```
(tell rpv set your $autonomous-attributes
      to (fuel-level position velocity))
```

(5) Supply an "**update your** *autonomous-attribute*" behavior for each autonomous attribute specified in (3). (These are contained in the simulation code in Appendix B).

(6) Place the name of all history-affecting attributes on the **$history-affecting-attributes** list for RPVs:

```
(tell rpv set your $history-affecting-attributes
 to (mpg speed trajectory))
```

We can now remove all explicit calls to "**update your** *attribute*" behaviors from our simulation code. The update-on-demand extensions will call these behaviors automatically when, and only when, it is necessary. Our control loop is reduced to the following:

```
(tell nclock when receiving (tick and update >n times)
      (loop for i from 1 to n
            do (tell !myself tick and update)))

(tell nclock when receiving (tick and update)
      (tell !myself tick)
      (loop for vehicle in (ask rpv recall your offspring)
            do (tell !vehicle check flight status)
               (tell !vehicle sense)))
```

Upon execution, autonomous attributes will be updated correctly. Figure 5.5 shows a record of the attribute values of Rpv2 for the simulation using update-on-demand processing. By comparing Figure 5.5 with Figure 5.2 (which has no automatic updating), we see that the history of fuel-level values in Figure 5.5 is correct. All computations were forced to be performed in the correct order (i.e., fuel-level was always updated before speed was changed), and fuel levels for the RPVs were not computed after they crashed. The values of the other autonomous attributes are still recomputed at each tick only because the simulation code references them. These artifactual processes are enabled simply by the initial declarations 1 through 6 listed above. Procedural calls for updating autonomous or history-affecting attributes are no longer necessary.

$stime	trajectory	speed	mpg	velocity	position	fuel-level	sensed-defenses	flight-status	plana
0	(1.0 1.0)	5.0	10.0	(3.5 3.5)	(0.0 3.0)	7.0	()	in-flight	()
1	-	-	-	(3.5 3.5)	(3.5 6.5)	6.5	(radar1)	-	((2 (react to sensed radar1)))
2	-	-	-	(3.5 3.5)	(7.0 10.0)	6.0	-	-	((3 (send warning communication)))
3	-	-	-	(3.5 3.5)	(10.6 13.6)	5.5	()	-	()
4	-	-	-	(3.5 3.5)	(14.1 17.1)	5.0	(radar2)	-	((5 (react to sensed radar2)))
5	-	-	-	(3.5 3.5)	(17.6 20.6)	4.5	()	-	((6 (receive communication m00002)) (6 (send warning communication)))
6	-	6.0	-	(3.5 3.5)	(21.2 24.2)	4.0	-	-	()
7	-	-	-	(4.2 4.2)	(25.4 28.4)	3.4	-	-	-
8	-	-	-	(4.2 4.2)	(29.6 32.6)	2.8	-	-	-
9	-	-	-	(4.2 4.2)	(33.9 36.9)	2.2	-	-	-
10	-	-	-	(4.2 4.2)	(38.1 41.1)	1.6	-	-	-
11	-	-	-	(4.2 4.2)	(42.4 45.4)	0.9	-	-	-
12	-	-	-	(4.2 4.2)	(46.6 49.6)	0.3	-	-	-
13	-	0.0	-	(4.2 4.2)	(50.9 53.9)	0.0	-	crashed	-
14	-	-	-	(0.0 0.0)	(50.9 53.9)	-	-	-	-
15	-	-	-	(0.0 0.0)	(50.9 53.9)	-	-	-	-

Fig. 5.5—History of attribute value changes for Rpv2 with update-on-demand processing

Although we have corrected our updating anomalies by using update-on-demand processing, we still face the problem that the behavior of the simulation depends directly on the simulation clock. Consider the outcome of the simulation when run with a ticksize of 2 rather than 1. Figure 5.6 shows attribute histories and plans for Rpv2 where the ticksize is 2. (The dashes indicate that the corresponding values were not actually computed for the corresponding times by Ross. This does not imply that the values of autonomous attributes would not have changed if they had been recalled.) Rpv2 does not start sensing Radar1 until time=2 and therefore the warning communication is not sent until time=3. We see here that the control loop forced Rpv2 to compute which objects it was sensing every 2 seconds, rather than every second (as in our previous examples). Thus, Rpv1 did not receive a warning message until one second later and did not speed up until later than it should have. Clearly the accuracy of the simulation is still tied directly to the ticksize of the simulation clock.

$stime	trajectory	speed	mpg	velocity	position	fuel-level	sensed-defenses	flight-status	plans
0	(1.0 1.0)	5.0	10.0	(3.5 3.5)	(0.0 3.0)	7.0	()	in-flight	()
1	-	-	-	-	-	-	-	-	-
2	-	-	-	(3.5 3.5)	(7.0 10.0)	6.0	(radar1)	-	((3 (react to sensed radar1)))
3	-	-	-	-	-	-	-	-	((4 (send warning communication)))
4	-	-	-	(3.5 3.5)	(14.1 17.1)	5.0	(radar2)	-	((5 (react to sensed radar2)))
5	-	-	-	-	-	-	-	-	((6 (send warning communication)))
6	-	-	-	(3.5 3.5)	(21.2 24.2)	4.0	()	-	((7 (receive communication m00002)))
7	-	6.0	-	(3.5 3.5)	(24.7 27.7)	3.5	-	-	-
8	-	-	-	(4.2 4.2)	(28.9 31.9)	2.9	-	-	-
9	-	-	-	-	-	-	-	-	-
10	-	-	-	(4.2 4.2)	(37.4 40.4)	1.7	-	-	-
11	-	-	-	-	-	-	-	-	-
12	-	-	-	(4.2 4.2)	(45.9 48.9)	0.5	-	-	-
13	-	-	-	-	-	-	-	-	-
14	-	0.0	-	(4.2 4.2)	(54.4 57.4)	0.0	-	crashed	-
15	-	-	-	-	-	-	-	-	-
16	-	-	-	(0.0 0.0)	(54.4 57.4)	-	-	-	-

Fig. 5.6—History of attribute value changes for Rpv2 with update-on-demand processing and simulation ticksize of 2

5.2. Demons

In the previous section, we removed some bookkeeping tasks from the simulation's control loop—namely, we eliminated explicit updating. However, we noted that we could not discard the loop completely because it was still responsible for "telling" RPVs when they were sensing other objects and when they had run out of fuel. In this section, we demonstrate the use of demons to handle these tasks. We show that by combining demons with update-on-demand processing, we can eliminate the need for any sort of control loop, thereby realizing a totally event-based approach to simulation processing.

Simulation demons monitor the state of the simulation and take specified actions if and when certain conditions exist. They provide a convenient mechanism for specifying non-intentional events, i.e., events that happen as side effects of other events; and they allow such specifications to be removed from the main simulation tasks. Events such as running out of fuel and crashing, or moving to within sensing range of an object, can be conveniently simulated through the use of demons.

The only tasks our control loop performs without demons are (1) monitoring each RPV's fuel level and notifying it when it runs out of fuel, and (2) monitoring each RPV's position to determine which objects it can sense and when. Therefore, we create one demon to monitor each RPV's fuel level and two demons to manage sensing: one will maintain initiation of sensing, and the other will control termination of sensing. The complete procedure a user must follow is given in Appendix A, and the complete code used to implement the demons we describe in this section is given in the demons.l file of Appendix B. Below we describe that code briefly.

To define a demon, we must supply the class of objects it monitors, which attributes it monitors, what other classes (if any) are involved, and four behaviors to indicate how the demon should function. Thus, to implement a demon to monitor an RPV's fuel level, we provide the following information. The class involved is the RPV class, and the attributes to be monitored are *fuel-level*, *mpg*, and *speed*. (In this scenario speed and rate of fuel consumption affect fuel level.) No other simulation classes are involved. We define the overall behavior of the demon in terms of the following four components. The *action-triggering condition* is specified as a fuel-level of zero and a speed greater than zero. The *action* component, commanding the RPV to crash, is executed when the action-triggering condition becomes true. The third demon component, a *continuation condition*, is a condition that must be met for the demon to be rescheduled. In this scenario, the demon need not continue monitoring the RPV's fuel level if it has landed; therefore, the continuation-condition is a speed greater than zero. Finally, the *time* at which the demon should "wake up" to check for the action-triggering-condition can be conveniently supplied in a function defined in terms of the RPV's current fuel level, speed, and rate of fuel consumption. When the demon is defined, it will first check for the the action-triggering-condition. If it succeeds, the demon will then execute the specified action. Otherwise, it will check the continuation-condition and will reschedule itself to "wake up" in the future if and only if that condition is satisfied. Note that if any of the monitored attributes are modified as the simulation runs, the demons monitoring those attributes will automatically reschedule themselves as necessary.

Demons designed for sensing are implemented in a slightly different fashion. Sensing involves two different simulation classes: a *sensing* class, and a *sensed* class. Therefore, we define a demon for the RPV (sensing) class with the Radar (sensed) entity as its secondary class. Because sensing involves multiple pairs of objects, a separate demon will be initiated for each pair of RPV and Radar objects. The attributes of RPVs that should

be monitored are speed and trajectory, because these attributes affect position, and sensing is defined in terms of distance between positions. The action-triggering-condition is fulfilled when the distance between an RPV and a Radar is less than the RPV's sensing-range. The resulting actions (performed by the RPV object) are (1) to add the Radar to its list of sensed-defenses, and (2) to plan to send a warning communication to its partner. The demon mechanism then invokes a sensing termination demon for this pair of objects. If the action-triggering-condition fails, the continuation-condition is evaluated. This condition is fulfilled if the RPV continues heading in the general direction of the Radar in question. The demon is then rescheduled for some time in the future. The rescheduling time is computed by a worst-case analysis of the time at which the RPV would start sensing the Radar if it were heading directly toward the radar.

The above discussion presents the mechanisms for *initiating* sensing between an RPV and a Radar. However, the concept of "being within sensing range" implies not a single event, but rather a continuous process. Therefore, once a Radar has been detected, another demon for *terminating* sensing must be invoked. The components corresponding to a termination demon, in essence, perform the reverse actions of the sensing initiation demon. Details of the four components of termination demons can be found in Appendix B.

Now that we have defined demons to handle the tasks previously performed in the simulation control loop, we can eliminate the loop completely and simply use the built-in Ross "**tick**" behavior:

```
(tell nclock tick n times)
```

Figure 5.7 shows value histories for attributes of Rpv2 when our example simulation was run with update-on-demand processing and demons. The *plans* attribute highlights the automatic scheduling of demons for monitoring fuel consumption, initiating sensing, and terminating sensing. As our example demonstrates, by incorporating update-on-demand processing and demons into the simulation, we have changed the modeling approach from a mixed event- and time-driven strategy to one that is strictly event-based. The simulation is controlled almost entirely by the scheduled plans of active objects, rather than the simulation clock. Therefore, changing the ticksize of the clock will not change the outcome of the simulation.

$stime	trajectory	speed	mpg	velocity	position	fuel-level	sensed-defenaes	flight-atatus	plans
0	(1.0 1.0)	5.0	10.0	(3.5 3.5)	(0.0 3.0)	7.0	()	in-flight	((0 (reschedule all out-of-fuel demons)) (0 (reschedule all sensing demons)))
1	-	-	-	(3.5 3.5)	(3.5 6.5)	-	(radar1)	-	((2 (react to sensed radar1)) (2 (activate not sensing radar1 demon)) (4 (activate sensing radar2 demon)) (14 (activate out of fuel demon)))
2	-	-	-	(3.5 3.5)	(7.0 10.0)	-	-	-	((3 (send warning communication)) (3 (activate not sensing radar1 demon)) (4 (activate sensing radar2 demon)) (14 (activate out of fuel demon)))
3	-	-	-	(3.5 3.5)	(10.6 13.6)	-	()	-	((4 (activate senaing radar2 demon)) (14 (activate out of fuel demon)))
4	-	-	-	(3.5 3.5)	(14.1 17.1)	-	(radar2)	-	((5 (react to sensed radar2)) (5 (activate not sensing radar2 demon)) (14 (activate out of fuel demon)))
5	-	-	-	(3.5 3.5)	(17.6 20.6)	-	()	-	((6 (send warning communication)) (6 (receive communication m00002)) (14 (activate out of fuel demon)))
6	-	6.0	-	(3.5 3.5)	(21.2 24.2)	4.0	-	-	((13 (activate out of fuel demon)))
7	-	-	-	-	-	-	-	-	-
8	-	-	-	-	-	-	-	-	-
9	-	-	-	-	-	-	-	-	-
10	-	-	-	-	-	-	-	-	-
11	-	-	-	-	-	-	-	-	-
12	-	-	-	-	-	-	-	-	-
13	-	0.0	-	(4.2 4.2)	(50.9 53.9)	0.0	-	crashed	()
14	-	-	-	-	-	-	-	-	-
15	-	-	-	-	-	-	-	-	-

Fig. 5.7—History of attribute value changes for Rpv2 with update-on-demand processing and simulation demons

5.3. Graphics-Delta Processing

We have just shown how utilization of update-on-demand processing and simulation demons enable us to eliminate artifactual programming tasks for consistency maintenance of time-dependent attributes, and for handling non-intentional asynchronous events. In this section we show how the *graphics-delta* approach we described in Section 4.3 eliminates the need for artifactual code to perform graphics tasks. In Section 5.3.1 we demonstrate methods for generating a graphics-delta frame at each update of the simulation clock. In Section 5.3.2. we show how to limit the graphical output to user-specified *graphically significant* events.

5.3.1. Output at Each Update of the Simulation Clock

The procedure a user must follow to include graphics-delta processing in a simulation is given in Section 3 of Appendix A. We extend our example simulation to generate a graphics-delta frame at each update by declaring which simulation objects are to be displayed graphically, and specifying how to display each object. To enable our simulation to produce graphics output, we add the class declarations described below. As we noted in Section 5.1, the step numbers listed below correspond to the steps outlined in Appendix A.3. Also keep in mind that the declarations below augment those "core" classes listed in Appendix B.1.

(1) Create all top-level classes as subclasses of the class **simulator**:

```
(tell simulator create new generic moving-object)
(tell simulator create new generic fixed-object)
(tell simulator create new generic communication)
```

(2) Specify which objects are to be displayed graphically by creating them as subclasses of the **monitored-object** class:

```
(tell monitored-object create generic rpv)
(tell monitored-object create generic radar)
(tell monitored-object create generic communication)
```

(Note: No step 3 is provided because we are not declaring any frame-triggering attributes as per Appendix A.3, step 3.) (4) Provide procedures for constructing primary and, optionally, secondary images of each object that is to be graphically displayed. We supply these procedures as values of the **$primary-image** and **$secondary-image** attributes:

```
(tell rpv set your $primary-image to
  (cond ((eq (ask !myself recall your flight-status) 'in-flight)
         (format *file*
                 "Time   A:  Display   A's   A icon at   A  %"
                 (ask nclock recall your $stime)
                 myself
                 (if (ask !myself recall your sensed-defenses)
                     then (ask !myself recall your sensing-color)
                     else (ask !myself recall your color))
                 (ask !myself recall your position)))
        ((eq (ask !myself recall your flight-status) 'crashed)
         (format *file*
                 "Time   A:  Display   A's explosion icon at   A  %"
                 (ask nclock recall your $stime)
                 myself
                 (ask !myself recall your position)))))
(tell rpv set your $secondary-image to
  (if (ask !myself recall your sensed-defenses) then
      (format *file*
       "Time   A:  Display   A's sensing-range icon of range   A  %"
              (ask nclock recall your $stime)
              myself
              (ask !myself recall your sensing-range))))
```

(The behaviors for constructing the images for the radar and communication type objects are given in the objects.l file in Section 2 of Appendix B.)

Figure 5.8 portrays the resulting object hierarchy. The new italicized classes are built into the Ross language to support declarative consistency maintenance. A detailed discussion describing implementation will be presented in Section 6. The bold-typed classes represent domain classes (as in Figure 5.1). Although the specific domain classes have not

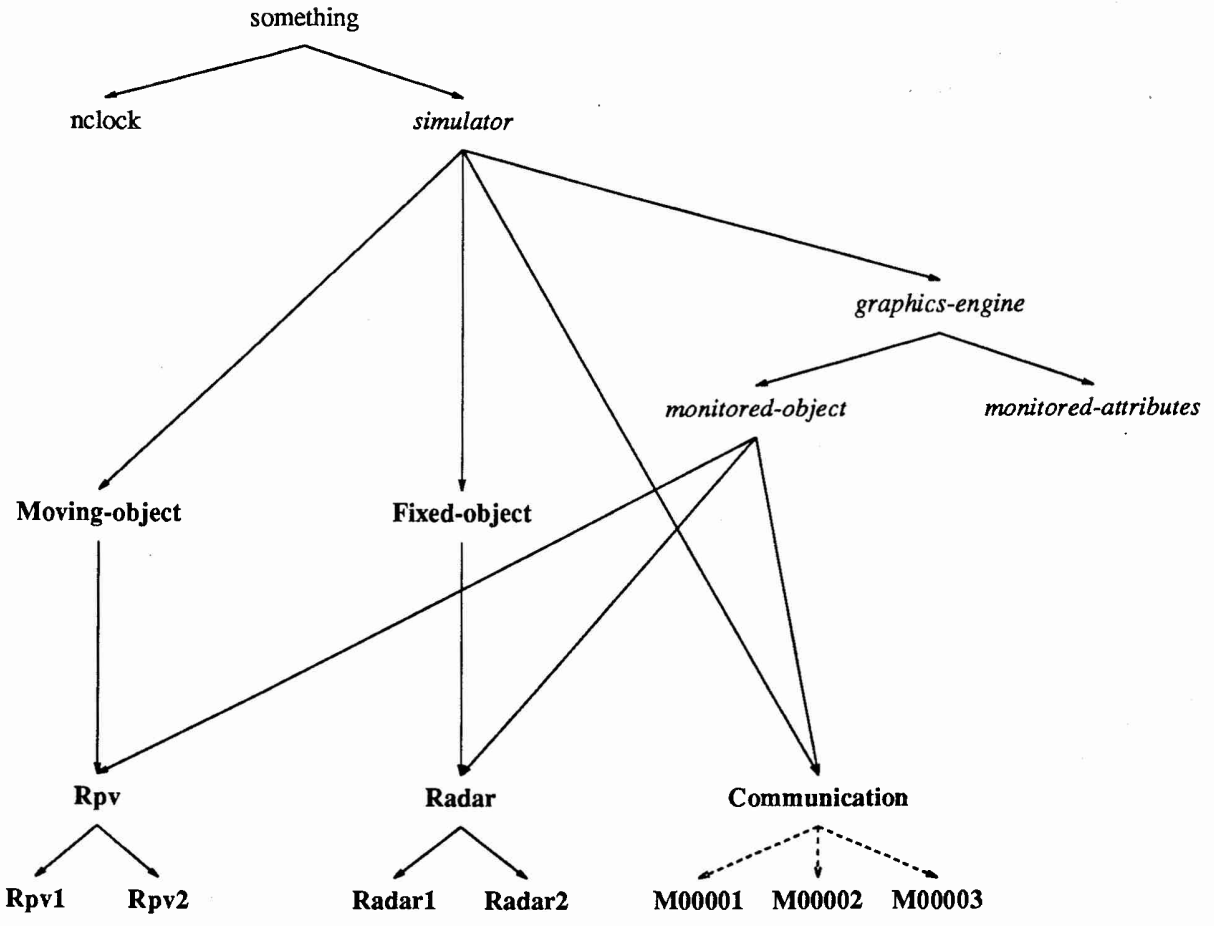

Fig. 5.8—Class and instance hierarchy for simulation in Ross with extensions

changed, they now inherit from both the simulator *and* monitored-object classes. Now we simply use the Ross "**gtick and display *n* times**" behavior to advance the simulation clock instead of the usual "**tick *n* times.**" The graphics-delta output produced by issuing

```
(tell nclock gtick and display 13 times)
```

is shown in Figure 5.9. We purposely chose to have our graphics output produced for this example in the form of character strings written to a file to emphasize that this facility is not dependent on specific graphics hardware or software packages. Further, it is beneficial to

have the option of generating text output in place of graphics images as this enables a simulation to be run from a terminal without graphics capabilities.

Figure 5.9 shows the graphics-delta frames produced throughout execution of the simulation. Note that the radar icons are displayed initially, but are not redisplayed after time 0. This situation is desirable because radars are static objects. Their positions do not change, and they do not have any time-dependent attributes; therefore, their graphical manifestations remain the same. RPVs, however, are dynamic objects. Until they run out of fuel, their positions change as the simulation progresses. Therefore, until they crash, their icons should be redisplayed at each update of the simulation clock. Warning communications are temporary objects. They only exist from the time they are sent until the time they are received. Consequently, they are only displayed while they are being transmitted. We have shown that Rpv1 crashes at time 9 and Rpv2 crashes at time 13. For simplicity in this example, the explosion icons are redisplayed for every graphics update following the explosion. The procedural aspect of the primary and secondary image attributes affords the flexibility to change an object's graphics representation. However, to display the explosion only once at time of impact would require a flag to be maintained with the object. This flag would indicate that the vehicle has been shot down and that the new graphics representation, an explosion, has already been displayed. In the next section, we show how *graphically significant* events, such as an explosion, are used control the display of graphics frames, thereby producing an explosion icon only at the time of impact.

5.3.2. Output for Graphically Significant Events

We have just shown how the graphics-delta processing facility can be used to generate a delta frame at each update of the simulation clock. Clearly, the frequency at which graphics-delta frames are generated depends on the increment size of the simulation clock. Suppose, however, that we are interested only in showing highlights of the simulation. That is, we wish to generate graphics output only when significant events take place, rather than at regular time intervals. Graphics-delta processing enables us to do this through specification of *graphically significant* events. The **$frame-triggering-attributes** list of a monitored-object is used for this purpose.

Suppose we wish to show the following events involving RPVs: initiation and termination of sensing, change in velocity, and crashing. At any given time, the objects that an RPV is sensing are contained in its sensed-defenses list. Therefore, whenever the value of an RPV's sensed-defenses list changes, we want a graphics frame to be generated. Similarly, we need to monitor the values of the velocity and flight-status attributes. The following declarations will produce the desired graphics output.

```
Time 0:  Display radar1's gray radar icon at (4.0 9.0)
Time 0:  Display radar2's gray radar icon at (16.0 16.0)
Time 0:  Display rpv1's blue icon at (5.0 2.0)
Time 0:  Display rpv2's blue icon at (0.0 3.0)
Time 1:  Display rpv1's blue icon at (8.5 5.5)
Time 1:  Display rpv2's purple icon at (3.5 6.5)
Time 1:  Display rpv2's sensing-range icon of range 4.0
Time 2:  Display rpv1's blue icon at (12.0 9.0)
Time 2:  Display rpv2's purple icon at (7.0 10.0)
Time 2:  Display rpv2's sensing-range icon of range 4.0
Time 3:  Display m00001's message icon from rpv2 to rpv1
Time 3:  Label m00001's icon 'speed-up'
Time 3:  Display rpv1's purple icon at (15.6 12.6)
Time 3:  Display rpv1's sensing-range icon of range 4.0
Time 3:  Display rpv2's blue icon at (10.6 13.6)
Time 4:  Display rpv1's purple icon at (19.1 16.1)
Time 4:  Display rpv1's sensing-range icon of range 4.0
Time 4:  Display rpv2's purple icon at (14.1 17.1)
Time 4:  Display rpv2's sensing-range icon of range 4.0
Time 5:  Display m00002's message icon from rpv1 to rpv2
Time 5:  Label m00002's icon 'speed-up'
Time 5:  Display rpv1's blue icon at (23.3 20.3)
Time 5:  Display rpv2's blue icon at (17.6 20.6)
Time 6:  Display m00003's message icon from rpv2 to rpv1
Time 6:  Label m00003's icon 'speed-up'
Time 6:  Display rpv1's blue icon at (27.6 24.6)
Time 6:  Display rpv2's blue icon at (21.2 24.2)
Time 7:  Display rpv1's blue icon at (31.8 28.8)
Time 7:  Display rpv2's blue icon at (25.4 28.4)
Time 8:  Display rpv1's blue icon at (36.8 33.8)
Time 8:  Display rpv2's blue icon at (29.6 32.6)
Time 9:  Display rpv1's explosion icon at (41.7 38.7)
Time 9:  Display rpv2's blue icon at (33.9 36.9)
Time 10:  Display rpv1's explosion icon at (41.7 38.7)
Time 10:  Display rpv2's blue icon at (38.1 41.1)
Time 11:  Display rpv1's explosion icon at (41.7 38.7)
Time 11:  Display rpv2's blue icon at (42.4 45.4)
Time 12:  Display rpv1's explosion icon at (41.7 38.7)
Time 12:  Display rpv2's blue icon at (46.6 49.6)
Time 13:  Display rpv1's explosion icon at (41.7 38.7)
Time 13:  Display rpv2's explosion icon at (50.9 53.9)
```

Fig. 5.9—Graphics-delta generated by advancing the simulation clock

```
(tell rpv set your $monitored-attributes
      to (flight-status sensed-defenses velocity))

(tell rpv set your $frame-triggering-attributes
      to (flight-status sensed-defenses velocity))
```

If we wish to display each communication as it is transmitted, we can declare its *content* attribute to be frame-triggering:

```
(tell communication set your $monitored-attributes
      to (content))

(tell communication set your $frame-triggering-attributes
      to (content))
```

If we supply the above declarations together with steps 1 through 4 from Section 5.3., we can use the "**gtick *n* times**" behavior to advance the simulation clock and generate graphics-delta frames only when graphically significant events occur. Thus,

```
(tell nclock gtick 13 times)
```

generates the delta output shown in Figure 5.10. Note that graphics frames are only generated for those ticks in which a graphically significant event has taken place. Note also that update-on-demand processing guarantees that autonomous attributes affecting the display of objects are brought up-to-date automatically when the object is displayed and their values are recalled. This ensures that the graphics display ill present the simulation in a consistent state.

In this example we have isolated artifactual procedural tasks and have replaced them with declarative specifications. This declarative methodology encourages a modular approach to programming; artifactual tasks can be localized and encapsulated. By removing detail from the top level, we have made the simulation code conceptually cleaner and therefore, easier to maintain and modify.

```
Time 0:  Display radar1's gray radar icon at (4.0 9.0)
Time 0:  Display radar2's gray radar icon at (16.0 16.0)
Time 0:  Display rpv1's blue icon at (5.0 2.0)
Time 0:  Display rpv2's blue icon at (0.0 3.0)
Time 1:  Display rpv1's blue icon at (8.5 5.5)
Time 1:  Display rpv2's purple icon at (3.5 6.5)
Time 1:  Display rpv2's sensing-range icon of range 4.0
Time 3:  Display m00001's message icon from rpv2 to rpv1
Time 3:  Label m00001's icon 'speed-up'
Time 3:  Display rpv1's purple icon at (15.6 12.6)
Time 3:  Display rpv1's sensing-range icon of range 4.0
Time 3:  Display rpv2's blue icon at (10.6 13.6)
Time 4:  Display rpv1's purple icon at (19.1 16.1)
Time 4:  Display rpv1's sensing-range icon of range 4.0
Time 4:  Display rpv2's purple icon at (14.1 17.1)
Time 4:  Display rpv2's sensing-range icon of range 4.0
Time 5:  Display m00002's message icon from rpv1 to rpv2
Time 5:  Label m00002's icon 'speed-up'
Time 5:  Display rpv1's blue icon at (23.3 20.3)
Time 5:  Display rpv2's blue icon at (17.6 20.6)
Time 6:  Display m00003's message icon from rpv2 to rpv1
Time 6:  Label m00003's icon 'speed-up'
Time 6:  Display rpv1's blue icon at (27.6 24.6)
Time 6:  Display rpv2's blue icon at (21.2 24.2)
Time 9:  Display rpv1's explosion icon at (41.7 38.7)
Time 9:  Display rpv2's blue icon at (33.9 36.9)
Time 13:  Display rpv2's explosion icon at (50.9 53.9)
```

Fig. 5.10—Graphics-delta output highlighting graphically significant events

6. DESIGN AND IMPLEMENTATION OF ROSS EXTENSIONS

We have extended the Ross language to maintain consistency within the state of a simulation, to handle execution of asynchronous events, and to maintain consistency in the graphical representation of a simulation. In implementing these extensions, our goal was to enable artifactual procedural computations to be removed from simulation code without requiring the simulation programmer to adopt a different programming style. In this section we describe the design and implementation of these extensions.

6.1. Update-on-Demand Processing

Our update-on-demand extensions maintain consistency among time-dependent attributes by bringing their values up-to-date when, and *only when*, necessary. These extensions update an autonomous attribute each time its value is requested but is out-of-date with respect to the current simulation time. They also update all of the autonomous attributes of an object before modifying any of its history-affecting attributes. We wanted these operations to be carried out automatically and transparently at the appropriate times during the execution of a simulation. However, we also wanted to make the usage of these extensions optional; we did not want to *require* the simulation programmer to utilize them. In object-oriented languages more sophisticated than Ross (such as Flavors), controlling the combination of ancestor behaviors can be easily performed by the simulation implementor. Allowing the coder to mix behaviors of ancestor objects would provide facilities that are both transparent and optional. However, Ross does not support such behavior combinations. Therefore, we were forced to add a new class, *simulator*, to the initial object hierarchy. The behaviors of *simulator* are a combination of those original behaviors of the *something* object and the new behaviors of the *monitored-object* class.

Figure 6.1 contains a representation of the new initial object hierarchy in Ross with consistency maintenance extensions. The italicized class names are those added to the initial hierarchy. The *simulator* class holds the new definitions of built-in Ross behaviors for setting, retrieving, and deleting attribute values. These new behaviors are inherited by all objects that are descendants of *simulator*. Thus, if users do not wish to utilize the consistency maintenance extensions, they can attach the hierarchy of simulation objects to *something*, as was done before the extensions were implemented.

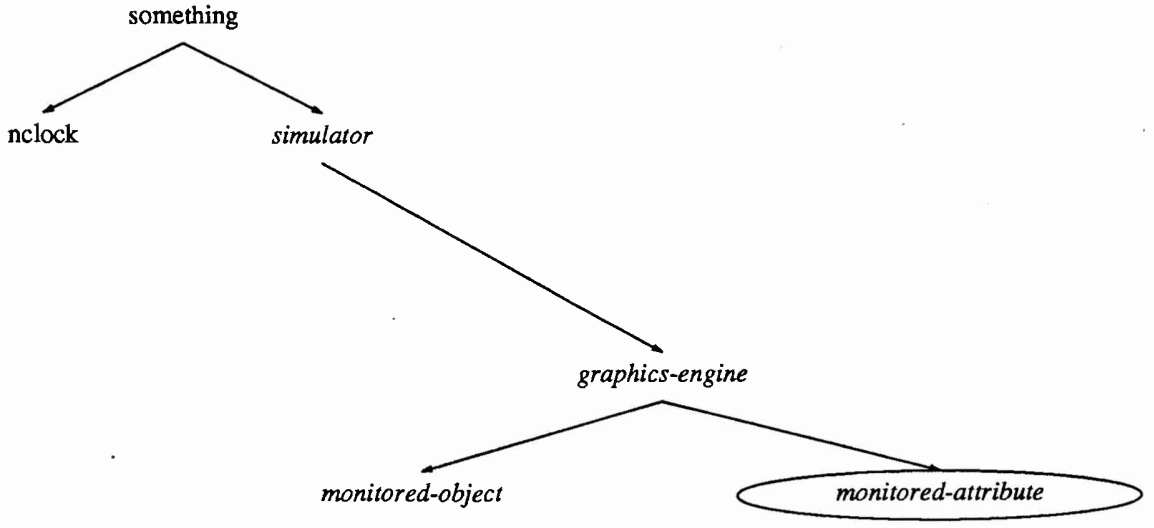

Fig. 6.1—Initial object hierarchy in ross with extensions

The *monitored-attribute* class is responsible for maintaining information about attributes of simulation objects. For example, it keeps a record of which attributes are autonomous, and which are history-affecting. If Ross allowed objects to have complex attributes—i.e., if attributes could be attached to attributes—we could have stored the information associated with a particular attribute where it belongs: with that particular attribute. Since this was not the case, we implemented our extensions to dynamically create new classes and instances under the *monitored-attribute* class to store this information. The oval in Figure 6.1 indicates that this class of objects and all bookkeeping operations performed on it are hidden from the user. Although the users are free to browse through the *monitored-attribute* class, they are not required to know of its existence.

6.2. Simulation Demons

Our demon facility handles execution of non-intentional events by implementing asynchronous demon processes. Instead of implementing demons as separate processes, we chose to implement them as behaviors that, when executed, check for the existence of the action-triggering condition and (if appropriate) reschedule themselves to be executed again sometime in the future. The execution of a demon behavior can be summarized as follows:

IF the *action-triggering condition* exists
  THEN execute the *action*
  ELSE IF the *continuation condition* exists
    THEN schedule this behavior to be executed *n* seconds from now,
      where *n* is the number of seconds computed by the prediction function

When a demon is defined initially, a behavior is automatically generated to perform these actions for the specified class(es) of objects. If a *simple* demon is being defined, each object in the corresponding class executes this behavior. If a *compound* demon is being defined, each object in the corresponding class executes this behavior once for each object in the corresponding secondary class.

Each demon behavior schedules itself to be re-executed until either the action-triggering condition exists and the corresponding action is triggered, or it appears that, given current conditions, the action-triggering condition will not exist in the future. In either case, it is possible that the state of the simulation may subsequently change, thereby bringing about a new possibility that the action-triggering condition may again arise. Therefore, all demon-affecting attributes are monitored.

We have redefined the built-in Ross "**set your** ..." behavior, which sets attribute values, to monitor demon-affecting attributes. Whenever the value of any of these attributes is modified, the "**set your** ..." behavior re-executes the demon behavior. The result of executing the demon behavior will either be (1) to execute the action, (2) to schedule execution of this behavior at some future time, or (3) to do nothing at all. In any case, if the behavior had previously been scheduled for execution at some future time, that time is now unscheduled.

Figure 6.2 contains sample code generated by the demon facility. In this example, a demon is defined to enable RPVs to sense Radars. Two *scheduling* behaviors are automatically generated with the demon behavior. The "**reschedule all sensing demons**" behavior is executed initially. It tells each object in the RPV class to execute the "**reschedule your sensing demons**" behavior. This behavior in turn tells each instance of the RPV class to execute the demon behavior (i.e., "**activate sensing ... demon**") for each instance of the Radar class. Speed and trajectory are declared to be demon-affecting attributes; therefore, an RPV is told to execute the "**reschedule your sensing demons**" whenever its speed or trajectory is changed.

User-supplied declaration:

```
(tell rpv define compound demon (sensing)
      with $secondary-class radar
           $demon-affecting-attributes (speed trajectory))
```

Generated behaviors:

```
;;  Scheduling Behaviors  ;;

 (tell rpv when receiving (reschedule all sensing demons)
      (loop for demon-object in (ask rpv find all instances)
            do (tell !demon-object reschedule your sensing demons)))

(tell rpv when receiving (reschedule your sensing demons)
      (loop for sobj in (ask radar find all instances)
            unless (equal sobj myself)
            do (tell !myself unplan (activate sensing !sobj demon))
               (tell !myself activate sensing !sobj demon)))

;;  Demon Behavior  ;;

(tell rpv when receiving (activate sensing >sobj demon)
      (if (ask !myself am i sensing !sobj)
          then (tell !myself i am sensing  !sobj)
          else (if (ask !myself shall i continue checking
                        if i am sensing !sobj)
                   then (tell !myself plan after
                              !(ask !myself compute how soon to check
                                    if i am sensing !sobj) seconds
                              activate sensing !sobj demon))))
```

Fig. 6.2—Demon declaration and corresponding behaviors generated by demon facility

6.3. Graphics-delta processing

The graphics-delta processing facility maintains a history of values of frame-triggering attributes. It generates a graphics-delta frame each time the simulation clock is about to be advanced, and when one or more frame-triggering attributes have been modified. We adopted the same implementation strategy for our graphics-delta extensions as we did for our update-on-demand extensions; we wanted the new operations necessary for graphics-delta processing to be executed transparently, and we wanted execution of these operations to be optional. Therefore, our graphics-delta extensions require usage of the new object hierarchy depicted in Figure 6.1.

Our update-on-demand extensions store new definitions of basic Ross behaviors in the *simulator* object. Similarly, our graphics-delta extensions utilize these new definitions to monitor changes in the value of frame-triggering attributes. Again, if users do not wish to utilize graphics-delta processing, they can hang the simulation object hierarchy from *something*, rather than from *simulator*. Our graphics-delta extensions also rely on the *monitored-attribute* class to maintain histories of values of frame-triggering attributes. All operations performed on instances of this class are still transparent to the user. However, the graphics-delta extensions also require an additional object to perform graphics tasks: the *graphics-engine* keeps track of whether or not a new graphics-delta frame must be generated at the end of processing for the current simulation time.

Each time the value of a frame-triggering attribute is modified, the *graphics-engine* sets its *update-display* flag (whose default value is false) to true. Then, when *nclock* finishes all processing for the current simulation time, the *graphics-engine* retrieves the value of its *update-display* flag. If it is true, the *graphics-engine* generates a new graphics-delta frame, and resets its *update-display* flag to false.

When the *graphics-engine* generates a graphics frame, it first brings all autonomous attributes of all objects up-to-date to force the simulation into a consistent state. It then loops through all displayable simulation objects and compares their most recently *displayed* attribute values against their most recently *computed* attribute values. Only those objects with differing displayed and computed attribute values are redisplayed. (The newly displayed values are then recorded as the most recently displayed values.) In this way, graphics-delta determines what graphical changes are necessary to produce a new graphics frame and generates *only* those changes.

Since Ross is an object-oriented language, we were able to implement our extensions to Ross in Ross itself, without making substantial changes to the underlying language. Therefore, users are not forced to utilize all or any of these extensions if they do not wish to. Furthermore, existing Ross simulations would *not* have to undergo major changes to utilize these extensions; they would only include additional declarations and remove artifactual code.

7. CONCLUSIONS AND FUTURE WORK

The product of our work has been the development of a methodology and support tools to aid the implementation of object-oriented simulations. We have developed a strategy for maintaining consistency among the entities of an object-oriented simulation. Maintaining consistency includes: (1) enforcing consistency within a given state of simulation processing, and (2) controlling a graphics display of a simulation to reflect, as accurately as possible, the simulation's processing. In this Note we have discussed three problematic implementation issues involved in consistency maintenance. We have provided a methodology in which declarative specifications of desired simulation behavior provide automatic consistency updating. In this section, we discuss some of the benefits and advantages of this approach, and conclude with some limitations and directions for future work.

7.1. Benefits and Advantages

One main advantage of our declarative facilities is the reduction of *artifactual* message scheduling and transmission. Artifactual messages are those events or activities that have no analogy in the world being modeled, but are necessary for simulating this world. Procedurally invoking update behaviors, projecting sensing and conflict events, and generating consistent graphics displays are distracting programming tasks.

A second advantage of automatic updating is a reduction in the amount of control code written by the user. As shown in Section 5, a simulation control block using a version of Ross without our simulation extensions must iteratively bring *up-to-date* all objects whose attribute values are related to time. Furthermore, *only* updating at fixed time increments in a control loop is not guaranteed to keep simulation values current. For those asynchronous activities that are being monitored, projection computations and triggering conditions must be included within control procedures.

Both of the previous benefits also suggest a third result: improved object-oriented programming style. The data abstraction and modularization techniques advocated by object-oriented programming languages cannot be employed effectively if programmers must maintain global autonomous and history-affecting attributes and invoke procedures for updating them. Also, changes or additions to autonomous, history-affecting, and frame-triggering attributes involve only a change in the declaration of objects and their attributes;

no perturbations to the actual behavior code is required. For instance, suppose a developer decides that an attribute, *x*, should be included as a frame-triggering attribute. Under the *display processor* strategy, this functionality would require some substantial additions to the simulation control loop to maintain the history of attribute *x*'s values. The display processor would be triggered during the control loop if the last displayed value of *x* has changed. In this situation, the code additions are localized within the simulation's control processing. If, however, the *incremental* graphics update strategy has been adopted, such a modification to graphics processing would require the programmer to search through all existing simulation code looking for places where attribute *x* is set, and make the appropriate additions to effect a new graphics frame. Using the described *graphics-delta* approach, attribute *x* needs only to be declared as a frame-triggering attribute, and the proper machinery is put into motion to monitor the value of attribute *x*, and additionally, produce a new graphics display frame when attribute *x* changes value.

Establishing a standard device-independent interface to the specific graphics drawing routines is another benefit of our approach to graphics updating. The Ross extensions enabling the graphics-delta approach do not make any assumptions about graphics output devices. Instead, we use the function-valued attributes, *primary-image* and *secondary-image*, as the slots that a programmer fills with device-dependent code or foreign function calls to invoke graphics display routines. Therefore, these *reserved* attributes are viewed as graphics hooks from which a programmer can hang graphics procedures. The built-in behavior requesting an object to *display* itself simply evaluates the primary-image and secondary-image attributes thereby producing the desired graphical picture.

Another improvement to graphics processing involves efficiency. Our simulation extensions monitor the history of frame-triggering objects and attributes; therefore, display of redundant or duplicate icons is eliminated. For instance, in the scenario described in Section 5, the radar objects are stationary in space. Our graphics facilities optimize the display processing by recognizing that a radar's position attribute remains the same throughout the simulation. Therefore, the display processor generates a radar icon only once, rather than redisplaying the radar object each time a new graphics frame is produced. Of course, some extra overhead is required for recording and checking the history of frame-triggering objects. Our initial investigations indicate that if the ratio of static to dynamic objects is high, then a time improvement is realized; however, future work is needed to measure these tradeoffs meaningfully in a realistic simulation.

7.2. Limitations and Future Work

During the development of this research, our purpose was to identify anomalies and inconsistencies that arise during the use of object-oriented languages for simulation processing. We initially itemized and documented these anomalies one by one, trying to understand the reason that the simulation drifted into an inconsistent state. As our analysis continued, it became apparent that many errors were caused by a fundamental problem in processing time-related attributes and dependencies. We identified three categories of problematic processing: autonomous attributes, asynchronous events, and graphics display interfaces. We strove to design the most general solution with the largest payoff; however, operating within limited resources forced us to impose some limitations on the extent of the solution's generality. Below we discuss four instances where we constrained our approach, the implications of those constraints, and future work intended to alleviate the constraints.

7.2.1. Localization of Second-Order Effects

One limitation we discussed earlier is the localization of autonomous and history-affecting attributes within a given object. Violating localization would correspond to a situation in which the position of one object depended on the speed (or other history-affecting attribute) of another object. In the current implementation of our Ross extensions, we solve second-order effects automatically by updating all autonomous attributes of an object immediately before any history-affecting attribute *of that object* is updated. In the absence of this localization assumption, handling second-order effects would require updating all autonomous attributes of *all* objects in the simulation. It is likely that most of the autonomous attributes are not history-dependent; therefore, the overhead of such computations would no longer be cost effective.

Our experience shows that it is relatively easy to avoid building models with dependencies of this kind. Of course, violating the localization assumption in special cases is allowable, provided the simulation performs the required updating explicitly; this requirement is no worse than our previous simulation systems, where the simulation performs *all* updating explicitly. Future efforts will consider more efficient and feasible techniques for maintaining non-localized second-order dependencies.

7.2.2. Cyclic Attribute Dependencies

Although the update-on-demand facility requires the simulation programmer to declare attribute dependencies, it does not prevent the formation of implicit updating loops. The programmer is therefore encouraged to construct an attribute dependency graph and maintain it throughout the simulation development. Future research will address removing this responsibility from the programmer.

7.2.3. Localization of Demon Monitoring Activities

The current demon implementation facilities require that attributes that may affect the behavior of a demon be limited to the attributes of one particular object. This is a reasonable assumption for many side effects that are triggered as a result of processing *only* that particular object. For example, a demon need only monitor an object's fuel level to determine whether it has run out of fuel. However, a demon that monitors the positions of two moving objects to determine whether or not they have collided needs to monitor the speed and trajectory values of *both* moving objects. Currently, monitoring the same attributes of multiple objects can be accomplished by creating a separate demon for each object. However, a demon implementation allowing monitoring of attributes across object boundaries would be more flexible and would improve the efficiency of demon monitoring activities.

7.2.4. Graphics Interface Supporting Animation

The use of *frame-triggering* attributes provides a declarative means for specifying those events that should produce an updated graphics display. The simulation control loop and graphics interface are configured to produce a display reflecting all frame-triggering events since the time of the previous display. Therefore, if more than one frame-triggering event has occurred at the same simulation time, the graphical realization of those events will be displayed simultaneously. However, under some conditions, it is desirable to generate separate graphic updates to visually sequence events that occur at the same simulation time. We refer to this technique as graphical *animation.*

Animation can be divided into several cases. A single frame-triggering attribute will rarely be changed by the simulator more than once during the same simulation time; if it is, it seems undesirable that this should produce more than one graphic update. However, setting two (or more) different frame-triggering attributes of an object during the same simulation time may make it appropriate to produce more than one graphic frame. For

example, an object might change its location and also send a communication to another object. To highlight these events, it may be desirable to show them separately. Similarly, setting the same or different attributes of different objects during the same simulation time might also enable animation, whereby each object's change produces its own graphics frame. Further specifying the semantics of animation and identifying conditions for enabling and disabling animation is necessary before designing animation capabilities. This analysis will help define the focus of our future work on animation facilities within the framework of the graphics-delta interface strategy.

In this Note we have presented a methodology for improving the design, development, and implementation of object-oriented simulations. Our initial goal was to reduce the level of effort required to model artifactual phenomena. However, we have observed that the techniques we have developed promote a more thorough learning and understanding of the semantics of applications, thereby resulting in more understandable and valid simulation systems.

Appendix A

PROCEDURES A USER MUST FOLLOW TO UTILIZE ROSS EXTENSIONS

A.1. Tasks a User Must Perform to Utilize Update-On-Demand Processing in a Simulation

(1) Create all top-level classes as subclasses of the class **simulator** rather than **something**. For example:

(tell simulator create new generic *class)*

(2) Identify every object that has one or more attributes whose current values are dependent (either directly or indirectly) on the passage of time. Each such object (or class of objects) must be created as a subclass of the special class **monitored-object**:

(tell monitored-object create generic *monitored-object-class)*

(3) Specify all of the object's attributes that are dependent on time. These must be placed on the **$monitored-attributes** list of the *monitored-object-class* given in 2:

(tell *monitored-object-class* **set your $monitored-attributes to(***attr1 attr2 ... attrn***))**

(4) Identify which attributes have continuous (as opposed to discrete) values and are dependent on time. Also include those attributes that must be brought up-to-date whenever any of the values they depend upon are changed. These must be placed on the **$autonomous-attributes** list:

(tell *monitored-object-class* **set your $autonomous-attributes**
to(*attr1 attr2 ... attrn***))**

Each of the attributes on the **$autonomous-attributes** list must also appear in the **$monitored-attributes** list in 3.

(5) Each autonomous attribute in the **$autonomous-attributes** list must have a corresponding behavior to bring its value up-to-date for the time at which it is called. This behavior will never be called directly from the user's simulation code. This must be defined as the "**update your** *autonomous-attribute*" behavior for the *monitored-object-class*:

(tell *monitored-object-class* **when receiving (update your** *autonomous-attribute*B)
code-to-compute-current-value-of-autonomous-attribute)

(6) Identify those attributes whose history of value changes affect the values of autonomous attributes. These must be placed on the **$history-affecting-attributes** list:

monitored-object-class **set your $history-affecting-attributes**
to (*attr1 attr2 ... attrn***))**

Each of the attributes on the **$history-affecting-attributes** list must also appear in the **$monitored-attributes** list in 3.

A.2. Tasks a User Must Perform to Define Demons in a Simulation

To define a simple demon:

(1) Create all top-level classes as subclasses of the class **simulator** rather than **something**. For example:

(tell simulator create new generic *class***)**

(2) Define the demon by specifying its name as a sequence of one or more atoms, the class of objects it monitors, and the attributes of the objects that are to be monitored as follows:

(tell *class* **define simple demon(***demon-name***)**
with $demon-affecting-attributes (*attr1 attr2 ... attrn***))**

(3) Provide a behavior to check for the existence of the condition the demon is watching for:

(tell *class* **when receiving (am i** *demon-name***)**
code-to-check-for-action-triggering-condition)

(4) Provide a behavior to take a certain action when the condition the demon is checking for exists:

(tell *class* **when receiving (i am** *demon-name*)
code-to-perform-demon-action)

(5) Provide a behavior to determine whether the condition being checked for still might exist sometime in the future:

(tell *class* **when receiving (shall i continue checking if i am** *demon-name)*
code-to-check-continuation-condition)

(6) Provide a behavior to compute how soon the demon should be rescheduled to check for the specified condition:

(tell *class* **when receiving (compute how soon to check if i am** *demon-name)*
code-to-compute-how-soon-to-recheck-for-action-triggering-condition)

To define a compound demon:

(1) Create all top-level classes as subclasses of the class **simulator** rather than **something**. For example:

(tell simulator create new generic *class*)

(2) Define the demon by specifying its name as a sequence of one or more atoms, the class of objects it monitors, the attributes of the objects that are to be monitored, and the secondary class of objects involved in the condition the demon is checking for:

(tell *class* **define compound demon** (*demon-name*)
with $secondary-class *secondary-class*
$demon-affecting-attributes (*attr1 attr2 ... attrn*))

(3) Provide a behavior to check for the existence of the condition between two objects the demon is watching for:

(tell *class* **when receiving (am i** *demon-name>secondary-object)*
code-to-check-for-action-triggering-condition)

(4) Provide a behavior to take a certain action when the condition the demon is checking for exists:

(tell *class* **when receiving (i am** *demon-name >secondary-object)*
code-to-perform-demon-action)

(5) Provide a behavior to determine whether the condition being checked for still might exist sometime in the future:

(tell *class* **when receiving (shall I continue checking if i am** *demon-name>secondary-object)*
code-to-check-continuation-condition)

(6) Provide a behavior to compute how soon the demon should be rescheduled to check for the specified condition:

(tell *class* **when receiving (compute how soon to check if i am** *demon-name>secondary-object)*
code-to-compute-how-soon-to-recheck-for-action-triggering-condition)

A.3. Tasks a User Must Perform to Utilize Graphics-Delta Processing in a Simulation

(1) Create all top-level classes as subclasses of the class **simulator** rather than **something.** For example:

(tell simulator create new generic *class*)

(2) Identify every object that is to be displayed graphically. Every such object (or class of objects) must be created as a subclass of the **monitored-object** class:

(tell monitored-object create generic *graphics-object-class*)

(3) Specify which of the objects' attributes should cause the graphics display to be updated each time their values are changed. These must be placed on both the **$monitored-attributes** and the **$frame-triggering-attributes** lists:

(tell *graphics-object-class* **set your $monitored-attributes**
to (*attr1 attr2 ... attrn***))**

(tell *graphics-object-class* **set your $frame-triggering-attributes**

to *attr1 attr2 ... attrn*))

(4) Provide behaviors for constructing primary and, optionally, secondary images of each object that is to be graphically displayed. These must be supplied as the values of the **$primary-image** and **$secondary-image** attributes, respectively:

(tell *graphics-object-class* **set your $primary-image**
to *code-to-display-primary-image-of-graphics-object*)

(tell *graphics-object-class* **set your $secondary-image**
to *code-to-display-secondary-image-of-monitored-object*)

Appendix B

ROSS CODE FOR EXAMPLE SIMULATION

In this appendix, we present the Ross code that implements the example simulation described in Section 5.

B.1. Ross Simulation Code

The original version of the Ross code implementing the example simulation is contained in four files: **control.l, objects.l, behaviors.l,** and **scenario.l.** The **control.l** file contains the main loop, which controls execution of the simulation; **objects.l** and **behaviors.l** contain specifications of classes of objects and corresponding behaviors; and **scenario.l** contains specifications of the instances in the particular scenario with which we chose to run this simulation.

control.l

```
;;  --Tick & update behaviors--  ;;

(tell nclock when receiving (tick and update >n times)
  (loop for i from 1 to n
        do (tell !myself tick and update)))

(tell nclock when receiving (tick and update)
  (tell !myself tick)
  (loop for vehicle in (ask rpv recall your offspring)
        do (tell !vehicle update your fuel-level)
           (tell !vehicle update your position)
           (tell !vehicle update your velocity)
           (tell !vehicle check flight status)
           (tell !vehicle sense)))
```

objects.l

```
(tell something create new generic moving-object)

(tell moving-object create new generic rpv with
  color           blue
  flight-status   in-flight
```

```
  fuel-level      nil
  fuel-level-time 0
  mpg             10.0
  partners        nil
  position        nil
  position-time   0
  sensed-defenses nil
  sensing-range   4.0
  sensing-color   purple
  speed           nil
  trajectory      nil
  velocity        nil)

(tell something create new generic fixed-object)

(tell fixed-object create new generic radar with
  color       gray
  position    nil)

(tell something create new generic communication with
  content  nil
  receiver nil
  sender   nil)
```

behaviors.l

```
;                                                    ;
;  If an RPV runs out of fuel in mid-air, it crashes  ;
;                                                    ;
(tell rpv when receiving (check flight status)
  (if (and (eq (ask !myself recall your flight-status) 'in-flight)
           (zerop (ask !myself recall your fuel-level)))
      then (tell !myself crash)))

;                                                                    ;
;  Update an RPV's list of defenses it is currently sensing          ;
;  If it detects an object not already in its list of sensed-defenses, ;
;  it sends a warning communication to each of its partners          ;
;                                                                    ;
(tell rpv when receiving (sense)
  (let ((sensing-range   (ask !myself recall your sensing-range))
        (sensed-defenses (ask !myself recall your sensed-defenses)))
       (loop for defense in (ask radar recall your offspring)
             do (if (lessp (ask !myself what is your distance to !defense)
                           sensing-range)
                    then (if (not (member defense sensed-defenses))
                             then (tell !myself add !defense to your list of
                                        sensed-defenses)
                                  (tell !myself plan after 1 seconds react to
                                        sensed !defense))
```

```
                    else (if (member defense sensed-defenses)
                             then (tell !myself remove !defense from your list
                                        of sensed-defenses))))))

;;  --Utility Behavior--  ;;

;                                          ;
;  Compute distance between two objects    ;
;                                          ;
(ask rpv when receiving (what is your distance to >object)
  (let (((x1 y1) (ask !myself recall your position))
        ((x2 y2) (ask !object recall your position)))
       (sqrt (plus (expt (difference x1 x2) 2) (expt (difference y1 y2) 2)))))

;;  --Attribute value update behaviors--  ;;

;                                                                          ;
;  Current velocity is a function of speed and trajectory                  ;
;  (xtraj, ytraj):                                                         ;
;                                                                          ;
;    ( speed/sqrt( xtraj**2 + ytraj**2 )*xtraj,                            ;
;      speed/sqrt( xtraj**2 + ytraj**2 )*ytraj )                           ;
;                                                                          ;
(tell rpv when receiving (update your velocity)
  (let ((speed         (ask !myself recall your speed)     )
        ((xtraj ytraj) (ask !myself recall your trajectory))
        q                                                  )
       (setq q (quotient speed (sqrt (plus (expt xtraj 2) (expt ytraj 2)))))
       (tell !myself
             set your velocity to !(list (times q xtraj) (times q ytraj)))))

;                                                                          ;
;  Current position is a function of previous position, (xpos, ypos);      ;
;  elapsed time since last position computation, deltat; and velocity,     ;
;  (xvel, yvel):                                                           ;
;                                                                          ;
(tell rpv when receiving (update your position)
  (let (((xpos ypos) (ask !myself recall your position)       )
        ((xvel yvel) (ask !myself recall your velocity)       )
        (deltat      (difference
                       (ask nclock recall your $stime)
                       (ask !myself recall your position-time))))
       (tell !myself set your position-time to !(ask nclock recall your $stime))
       (tell !myself set your position to
             !(list (plus xpos (times deltat xvel))
                    (plus ypos (times deltat yvel))))))

;                                                                          ;
;  Current fuel level is a function of previous fuel level, prevlevel;     ;
;  elapsed time since last fuel level computation, deltat; speed; and      ;
;  rate of fuel consumption, mpg:                                          ;
```

```
;                                                                    ;
(tell rpv when receiving (update your fuel-level)
  (let ((prevlevel (ask !myself recall your fuel-level)       )
        (speed     (ask !myself recall your speed)            )
        (mpg       (ask !myself recall your mpg)              )
        (deltat    (difference
                    (ask nclock recall your $stime)
                    (ask !myself recall your fuel-level-time))))
       (tell !myself
             set your fuel-level-time to !(ask nclock recall your $stime))
       (tell !myself set your fuel-level to
             !(max (difference prevlevel (times deltat (quotient speed mpg)))
                   0))))

;;  --Sensing & communication transmission behaviors--  ;;

;                                                                    ;
;  When a defense is sensed, a warning communication is scheduled to  ;
;  be sent                                                           ;
;                                                                    ;
(tell rpv when receiving (react to sensed >defense)
  (tell !myself plan after 1 seconds send warning communication))

;                              ;
;  Create a warning message    ;
;                              ;
(tell rpv when receiving (send warning communication)
  (loop for partner in (ask !myself recall your partners)
        do (let ((message (ask communication
                               create new instance !(intern (gensym 'm)) with
                                   sender   !myself
                                   receiver !partner
                                   content  speed-up)))
                (tell !message get delivered))))

;                                      ;
;  Schedule a message to be received   ;
;                                      ;
(tell communication when receiving (get delivered)
  (let ((recipient (ask !myself recall your receiver)))
       (tell !recipient plan after 1 seconds receive communication !myself)))

;                                                          ;
;  Recipient takes some action based on message content    ;
;                                                          ;
(tell rpv when receiving (receive communication >message)
  (if (eq (ask !message recall your content) 'speed-up)
      then (tell !myself increment your speed by 1))
  (tell !message kill yourself))

;;  --Crash behavior--  ;;
```

```
;                                                                          ;
;  A vehicle which crashes is no longer considered to be a partner by     ;
;  each of its partners.  It stops moving, so its speed is set to zero.;
;                                                                          ;
(tell rpv when receiving (crash)
  (tell !myself set your speed to 0)
  (loop for partner in (ask !myself recall your partners)
        do (tell !partner remove !myself from your list of partners))
  (tell !myself forget your partners)
  (tell !myself set your flight-status to crashed))
```

scenario.l

```
(tell rpv create new instance rpv1 with
  fuel-level      5.0
  partners        (rpv2)
  position        (5.0 2.0)
  sensed-defenses ()
  speed           5.0
  trajectory      (1.0 1.0)
  velocity        !(list (quotient 5.0 (sqrt 2.0))
                         (quotient 5.0 (sqrt 2.0))))

(tell rpv create new instance rpv2 with
  fuel-level      7.0
  partners        (rpv1)
  position        (0.0 3.0)
  sensed-defenses ()
  speed           5.0
  trajectory      (1.0 1.0)
  velocity        !(list (quotient 5.0 (sqrt 2.0))
                         (quotient 5.0 (sqrt 2.0))))



(tell radar create new instance radar1 with
  position    (4.0 9.0))

(tell radar create new instance radar2 with
  position    (16.0 16.0))
```

B.2. Ross Simulation Code Utilizing Update-on-Demand, Demon, and Graphics-Delta Extensions

The Ross code implementing the example simulation utilizing update-on-demand processing, simulation demons, and graphics-delta processing extensions is contained in four

files: **objects.l, behaviors.l, scenario.l** and **demons.l.** The files **objects.l** and **behaviors.l** contain specifications of classes of objects and corresponding behaviors; **scenario.l** contains specifications of the instances in the particular scenario with which we chose to run this simulation; and **demons.l** contains definitions of demons for handling sensing and running out of fuel.

objects.l

```
(tell simulator create new generic moving-object)

(tell moving-object create new generic rpv with
  color           blue
  flight-status   in-flight
  fuel-level      nil
  fuel-level-time 0
  mpg             10.0
  partners        nil
  position        nil
  position-time   0
  sensed-defenses nil
  sensing-range   4.0
  sensing-color   purple
  speed           nil
  trajectory      nil
  velocity        nil)

(tell monitored-object create generic rpv with
  $monitored-attributes
      (flight-status fuel-level mpg position sensed-defenses speed trajectory
       velocity)
  $autonomous-attributes
      (fuel-level position velocity)
  $frame-triggering-attributes
      (flight-status sensed-defenses velocity)
  $history-affecting-attributes
      (mpg speed trajectory)
  $primary-image
      (cond ((eq (ask !myself recall your flight-status) 'in-flight)
             (format *file*
                     "Time A:  Display A's A icon at A%"
                     (ask nclock recall your $stime)
                     myself
                     (if (ask !myself recall your sensed-defenses)
                         then (ask !myself recall your sensing-color)
                         else (ask !myself recall your color))
                     (ask !myself recall your position)))
            ((eq (ask !myself recall your flight-status) 'crashed)
             (format *file*
```

```
                    "Time A:  Display A's explosion icon at A%"
                    (ask nclock recall your $stime)
                    myself
                    (ask !myself recall your position))))
  $secondary-image
      (if (ask !myself recall your sensed-defenses)
          then (format *file*
                    "Time A:  Display A's sensing-range icon of range A%"
                    (ask nclock recall your $stime)
                    myself
                    (ask !myself recall your sensing-range))))

(tell simulator create new generic fixed-object)

(tell fixed-object create new generic radar with
  color       gray
  position    nil)

(tell monitored-object create generic radar with
  $primary-image
      (format *file* "Time A:  Display A's A radar icon at A%"
              (ask nclock recall your $stime)
              myself
              (ask !myself recall your color)
              (ask !myself recall your position)))


(tell simulator create new generic communication with
  content  nil
  receiver nil
  sender   nil)

(tell monitored-object create generic communication with
  $monitored-attributes
      (content)
  $frame-triggering-attributes
      (content)
  $primary-image
      (format *file* "Time A:  Display A's message icon from A to A%"
              (ask nclock recall your $stime)
              myself
              (ask !myself recall your sender)
              (ask !myself recall your receiver))
  $secondary-image
      (format *file* "Time A:  Label A's icon 'A'%"
              (ask nclock recall your $stime)
              myself
              (ask !myself recall your content)))
```

behaviors.l

```
;;  --Attribute value update behaviors--  ;;

;                                                                    ;
;  Current velocity is a function of speed and trajectory            ;
;  (xtraj, ytraj):                                                   ;
;                                                                    ;
;    ( speed/sqrt( xtraj**2 + ytraj**2 )*xtraj,                      ;
;      speed/sqrt( xtraj**2 + ytraj**2 )*ytraj )                     ;
;                                                                    ;
(tell rpv when receiving (update your velocity)
  (let ((speed          (ask !myself recall your speed)      )
        ((xtraj ytraj) (ask !myself recall your trajectory))
        q                                                    )
       (setq q (quotient speed (sqrt (plus (expt xtraj 2) (expt ytraj 2)))))
       (tell !myself set your velocity
             to !(list (times q xtraj) (times q ytraj)))))

;                                                                                  ;
;  Current position is a function of previous position, (xpos, ypos);              ;
;  elapsed time since last position computation, deltat; and velocity,             ;
;  (xvel, yvel):                                                                   ;
;                                                                                  ;
(tell rpv when receiving (update your position)
  (let (((xpos ypos) (ask !myself recall your position)       )
        ((xvel yvel) (ask !myself recall your velocity)       )
        (deltat      (difference
                       (ask nclock recall your $stime)
                       (ask !myself recall your position-time))))
       (tell !myself set your position-time to !(ask nclock recall your $stime))
       (tell !myself set your position to
             !(list (plus xpos (times deltat xvel))
                    (plus ypos (times deltat yvel))))))

;                                                                                  ;
;  Current fuel level is a function of previous fuel level, prevlevel;             ;
;  elapsed time since last fuel level computation, deltat; speed; and              ;
;  rate of fuel consumption, mpg:                                                  ;
;                                                                                  ;
(tell rpv when receiving (update your fuel-level)
  (let ((prevlevel (ask !myself recall your fuel-level)       )
        (speed     (ask !myself recall your speed)            )
        (mpg       (ask !myself recall your mpg)              )
        (deltat    (difference
                     (ask nclock recall your $stime)
                     (ask !myself recall your fuel-level-time))))
       (tell !myself set your fuel-level-time to
             !(ask nclock recall your $stime))
       (tell !myself set your fuel-level to
             !(max (difference prevlevel (times deltat (quotient speed mpg)))
                   0))))
```

```
;;  --Sensing & communication transmission behaviors--  ;;

;                                                                    ;
;  When a defense is sensed, a warning communication is scheduled to ;
;  be sent                                                           ;
;                                                                    ;
(tell rpv when receiving (react to sensed >defense)
  (tell !myself plan after 1 seconds send warning communication))

;                              ;
;  Create a warning message    ;
;                              ;
(tell rpv when receiving (send warning communication)
  (loop for partner in (ask !myself recall your partners)
        do (let ((message (ask communication
                                create new instance !(intern (gensym 'm)) with
                                    sender   !myself
                                    receiver !partner
                                    content  speed-up)))
                (tell !message get delivered))))

;                                    ;
;  Schedule a message to be received ;
;                                    ;
(tell communication when receiving (get delivered)
  (let ((recipient (ask !myself recall your receiver)))
       (tell !recipient plan after 1 seconds receive communication !myself)))

;                                                          ;
;  Recipient takes some action based on message content    ;
;                                                          ;
(tell rpv when receiving (receive communication >message)
  (if (eq (ask !message recall your content) 'speed-up)
      then (tell !myself increment your speed by 1))
  (tell !message kill yourself))


;;  --Crash behavior--  ;;


;                                                                    ;
;  A vehicle which crashes is no longer considered to be a partner by ;
;  each of its partners.  It stops moving, so its speed is set to zero.;
;                                                                    ;
(tell rpv when receiving (crash)
  (tell !myself set your speed to 0)
  (loop for partner in (ask !myself recall your partners)
        do (tell !partner remove !myself from your list of partners))
  (tell !myself forget your partners)
  (tell !myself set your flight-status to crashed))
```

scenario.l

```
(tell rpv create new instance rpv1 with
  fuel-level      5.0
  partners        (rpv2)
  position        (5.0 2.0)
  sensed-defenses ()
  speed           5.0
  trajectory      (1.0 1.0)
  velocity        !(list (quotient 5.0 (sqrt 2.0))
                         (quotient 5.0 (sqrt 2.0))))

(tell rpv create new instance rpv2 with
  fuel-level      7.0
  partners        (rpv1)
  position        (0.0 3.0)
  sensed-defenses ()
  speed           5.0
  trajectory      (1.0 1.0)
  velocity        !(list (quotient 5.0 (sqrt 2.0))
                         (quotient 5.0 (sqrt 2.0))))

(tell radar create new instance radar1 with
  position    (4.0 9.0))

(tell radar create new instance radar2 with
  position    (16.0 16.0))
```

demons.l

```
;;                          ;;
;;  --out-of-fuel demon--   ;;
;;                          ;;


(tell rpv define simple demon (out of fuel) with
  $demon-affecting-attributes (fuel-level mpg speed))

;                              ;
;  out-of-fuel demon behaviors ;
;                              ;

(ask rpv when receiving (am i out of fuel)
  (let ((level (ask !myself recall your fuel-level)))
       (and level (not (plusp level)))))

(tell rpv when receiving (i am out of fuel)
  (tell !myself crash))

(ask rpv when receiving (shall i continue checking if i am out of fuel)
```

```
  (let ((speed (ask !myself recall your speed)))
       (and speed (plusp speed))))

(ask rpv when receiving (compute how soon to check if i am out of fuel)
  (let ((mpg (ask !myself recall your mpg))
        (level (ask !myself recall your fuel-level))
        (speed (ask !myself recall your speed)))
       (if (and mpg level speed)
           then (max (fix (plus (quotient (times mpg level) speed) 0.999)) 1)
           else 1)))

;;                        ;;
;;  --sensing demon--     ;;
;;                        ;;

(tell rpv define compound demon (sensing) with
  $secondary-class             radar
  $demon-affecting-attributes (speed trajectory))

;                             ;
;  sensing demon behaviors    ;
;                             ;

(ask rpv when receiving (am i sensing >obj)
  (<= (ask !myself what is your distance to !obj)
      (ask !myself recall your sensing-range)))

(tell rpv when receiving (i am sensing >obj)
  (if (not (memq obj (ask !myself recall your sensed-defenses)))
      then (tell !myself add !obj to your list of sensed-defenses)
        .  (tell !myself plan after 1 seconds react to sensed !obj)
           (tell !myself activate not sensing !obj demon)))

(ask rpv when receiving (shall i continue checking if i am sensing >obj)
  (let (((xtraj ytraj) (ask !myself recall your trajectory))
        (speed         (ask !myself recall your speed))
        ((xpos ypos)   (ask !myself recall your position))
        ((xlpos ylpos) (ask !obj recall your position)))
       (and (plusp (plus (times xtraj (difference xlpos xpos))
                         (times ytraj (difference ylpos ypos))))
            (plusp speed))))

(ask rpv when receiving (compute how soon to check if i am sensing >obj)
  (max (fix (plus (quotient (difference
                              (ask !myself what is your distance to !obj)
                              (ask !myself recall your sensing-range))
                            (ask !myself recall your speed))
                  0.999))
       1))

;;                          ;;
```

```
;;  --not-sensing demon--  ;;
;;                         ;;

(tell rpv define compound demon (not sensing) with
  $secondary-class            radar
  $demon-affecting-attributes ())

;                                ;
;  not-sensing demon behaviors   ;
;                                ;

(ask rpv when receiving (am i not sensing >obj)
  (> (ask !myself what is your distance to !obj)
     (ask !myself recall your sensing-range)))

(tell rpv when receiving (i am not sensing >obj)
  (tell !myself remove !obj from your list of sensed-defenses))

(ask rpv when receiving (shall i continue checking if i am not sensing >obj)
  t)

(ask rpv when receiving (compute how soon to check if i am not sensing >obj)
  (ask nclock recall your $ticksize))

;;  --utility behaviors--  ;;

(ask simulator when receiving (what is your distance to >object)
  (let (((x1 y1) (ask !myself recall your position))
        ((x2 y2) (ask !object recall your position)))
       (sqrt (plus (expt (difference x1 x2) 2)
                   (expt (difference y1 y2) 2)))))
```

REFERENCES

1. Birtwistle, G., Dahl, O., Myhrhaug, B., and Nygaard. K., *Simula Begin,* Van Nostrand Reinhold, New York, 1973.

2. Borning, A., "Thinglab: A Constraint-Oriented Simulation Laboratory," Doctoral Dissertation, Stanford University, 1979.

3. Callero, M., Veit, C., and Rose, B., "Combat Identification and Fratricide: SHORAD Preliminary Findings," The RAND Corporation, N-2403-A, December 1985.

4. Conker, R.S., Davidson, J.R., Groverston, R.K., and Nugent, R.O., "The Battlefield Environment Model," MTR-83W00245, The Mitre Corporation, McLean, VA, 1983.

5. Dockery, J.T., "Structure of Command and Control Analysis," in *Proceedings of the Symposium on Modeling and Analysis of Defense Processes,* Brussels, 1982.

6. Elzas, M.S., "The Applicability of Artificial Intelligence Techniques to Knowledge Representation in Modelling and Simulation," pp. 19-40 in *Modelling and Simulation Methodology in the Artificial Intelligence Era,* (ed.), M.S. Elzas et al., Elsevier Science Publishers B.V., Amsterdam, 1986.

7. Goldberg, A. and Robson, D., *Smalltalk-80: The Language and Its Implementation,* Addison-Wesley Publishing Company, Menlo Park, CA, 1982.

8. "Graphics Kernel System: Functional Description," ANSI X3, 124-1885, American National Standard for Information System, 1985.

9. Hill, T.R. and Roberts, S.D., "A Prototype Knowledge-Based Simulation Support System," *Simulation* **48**(4), pp. 152-161, April 1987.

10. Hilton, M., "ERIC User's Manual," Internal Memo, RADC/COES, Rome, NY, 1986.

11. Kahn, K.M., "Director Guide," AI Memo 482B, Massachusetts Institute of Technology, Cambridge, MA, 1979.

12. Klahr, P., McArthur, D., and Narain, S., "SWIRL: An Object-Oriented Air Battle Simulator," pp. 331-334 in *Proceedings of the Second National Conference on Artificial Intelligence,* Pittsburgh, PA, 1982.

13. Klahr, P., McArthur, D., Narain, S. and Best, E., "SWIRL: Simulating Warfare in the ROSS Language," The RAND Corporation, N-1885-AF, September 1982.

14. Klahr, P., Ellis, J. Jr., Giarla, W., Narain, S., Cesar, E. Jr., and Turner, S., "TWIRL: Tactical Warfare in the ROSS Language," The RAND Corporation, R-3158-AF, October 1984.

15. Kornell, J., "Reflections On Using Knowledged Based Systems for Military Simulation," *Simulation* **48**(4), pp. 144-148, April 1987.

16. McArthur, D., Klahr, P., and Narain, S., "ROSS: An Object-Oriented Language for Constructing Simulations," The RAND Corporation, R-3160-AF, 1984.

17. McArthur, D., Klahr, P., and Narain, S., "The ROSS Language Manual," The RAND Corporation, N-1854-1-AF, September 1985.

18. McArthur, D., "Extending Expert Systems to be Expert Tutoring Aids," The RAND Corporation, R-3443-ARPA, 1987.

19. Newman, N.M. and Dam, A. Van, "Recent Efforts Toward Graphics Standardization," *ACM Computing Surveys* **10**(4), pp. 365-380, December 1978.

20. Nugent, R.O., "A Preliminary Evaluation of Object-Oriented Programming for Ground Combat Modeling," WP-83W00407, The Mitre Corporation, McLean, VA, 1983.

21. Overstreet, C.M. and Nance, R.E., "A Specification Language to Assist in Analysis of Discrete Event Simulation Models," *Communications of the ACM* **28**(2), pp. 190-210, February 1985.

22. Radiya, A. and Sargent, R.G., "ROBS: A Knowledge-Based Approach to Simulation," in *Proceedings of the 4th International Symposium on Modelling and Simulation Methodology*, Tucson, AZ.

23. Reddy, R., "Epistemology of Knowledge Based Simulation," *Simulation* **48**(4), pp. 162-166, April 1987.

24. Rothenberg, J., "Object-Oriented Simulation: Where Do We Go from Here?," pp. 464-469 in *Proceedings of 1986 Winter Simulation Conference*, Washington, DC, December 1986.

25. Rothenberg, J., "Hoses for Calling Functions from Lisp," The RAND Corporation, N-2546-ARPA, (unpublished document).

26. Rothenberg, J., Steeb, R., Shapiro, N., Narain, S., Cammarata, S., Gates, B., Florman, B., Hefley, C., Bankes, S., and Kameny, I., "Knowledge- Based Simulation: An Interim Report," The RAND Corporation, N-2543-ARPA, (unpublished document).

27. Ruiz-Mier, S. and Talavage, J., "A Hybrid Paradigm for Modeling of Complex Systems," *Simulation* **48**(4), pp. 135-141, April 1987.

28. Steeb, R., Cammarata, S., Narain, S., Rothenberg, J., and Giarla, W., "Cooperative Intelligence for Remotely Piloted Vehicle Fleet Control: Analysis and Simulation," The RAND Corporation, R-3408-ARPA, 1986.

29. Steeb, R., McArthur, D., Cammarata, S., Narain, S., and Giarla, W., "Distributed Problem Solving for Air Fleet Control: Framework and Implementation," pp. 391-432 in *Expert Systems: Techniques, Tools and Applications*, ed. P. Klahr et al., Addison-Wesley Publishing Company, Menlo Park, CA, 1986.

30. Stefik, M.J., Bobrow, D.G., and Kahn, K.M., "Integrating Access- Oriented Programming into a Multiparadigm Environment," *IEEE Software*, pp, 10-18, January 1986.

31. Stefik, M. and Bobrow, D., "Object-Oriented Programming: Themes and Variations," *AI Magazine* **6**(4), pp. 40-62, 1986.